Design and Analysis of Fault-Tolerant Distributed Real-Time Computer Systems

Final Report
Contract No. N00014-87-K-0231

July 25 , 1991

Project Director : K. H. Kim

Contractor : The Regents of the University of California
Dept. of Electrical & Computer Engineering
University of California
Irvine, California 92717

Sponsor : Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217-5000

Project Monitor : Dr. Gary M. Koob, Code 1133

"Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the US Government.

Table of Contents

Section

1. Introduction
2. Research Directions
3. Summary of the Phase-I Research Conducted at USF
4. Results of the Phase-II Research Conducted at UCI
5. Conclusion

Appendix A

A.I Designing Fault Tolerance Capabilities into Real-Time Distributed Computer Systems

A.II Approaches for System-Level Fault Tolerance in Distributed Real-Time Computer Systems

A.III Approaches to Implementation of a Repairable Distributed Recovery Block Scheme

A.IV Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications

A.V Performance Impacts of Look-Ahead Execution in the Conversation Scheme

A.VI Implementation of the Conversation Scheme in Loosely Coupled Distributed Computer Systems

A.VII Implementation of the Conversation Scheme in Message-Based Distributed Computer Systems

A.VIII Programmer-Transparent Coordination of Recovering Parallel Processes: Philosophy and Rules for Efficient Implementation

A.IX Issues in Design of Temporary Blackout Handling Capabilities into Tightly Coupled Computer Networks

A.X An Approach to Experimental Evaluation of Real-Time Fault-Tolerant Distributed Computing Schemes

A.XI Consistency Constraints in Distributed Real Time Systems

A.XII Diagnosabilities of Hypercubes under the Pessimistic One-Step Diagnosis Strategy

A.XIII Real-Time Distributed Computing Testbeds Established

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION The Regents of the University of California 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research Resident Representative
6c. ADDRESS (City, State, and ZIP Code) Dept. of Electrical & Computer Engineering Univ. of California Irvine, CA 92717	7b. ADDRESS (City, State, and ZIP Code) Univ. of California, San Diego Scripps Inst. of Oceanography, A-034 La Jolla, CA 92093-0234
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Office of Naval Research 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-87-K-0231
8c. ADDRESS (City, State, and ZIP Code) Code 1133 800 North Quincy Street Arlington, VA 22217-5000	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Design and Analysis of Fault-Tolerant Distributed Real-Time Computer Systems	
12. PERSONAL AUTHOR(S) K.H. Kim	
13a. TYPE OF REPORT Final Technical 13b. TIME COVERED FROM 87/03/01 TO 89/12/31	14. DATE OF REPORT (Year, Month, Day) 1991, July, 25 15. PAGE COUNT 193
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
real time computer, fault tolerance, distributed computing, recovery, temporal behavior

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The objective of this project was to contribute to the establishment of the scientific foundation for designing fault-tolerant distributed computer systems. Main results obtained from this research project are as follows.

(1) Identification of critical research issues and some promising research directions in real-time fault-tolerant distributed computing,

(2) A skeleton of the foundation for realizing system-level fault tolerance, which includes among others the DRB (distributed recovery block) scheme, the DCONV (distributed conversation) scheme, the PTC (programmer-transparent coordination) scheme, a TB (temporary blackout) handling scheme, and the complementary relationship among the schemes; These schemes enable the computer system to detect and recover from both hardware and software faults without missing the deadlines for processing important data and delivering outputs to the controlled object/environment,

(3) A preliminary structure of a model of real-time distributed computation,

--- continue on reverse ---

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL K.H. Kim	22b. TELEPHONE (Include Area Code) (714)856-5542 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Unclassified

Item 19. (cont.)

(4) A theoretical investigation into the efficiency and diagnostic power of basic processor-level diagnosis approaches in diagnosing hypercubes conducted,
(5) An enhancement of three of the real-time computer network testbeds established in the UCI DREAM (Distributed Real-time Ever Available Microcomputing) Laboratory made.

1. Introduction

This report summarizes the main results of our research carried out at the University of California, Irvine (UCI) under Contract No. N00014-87-K-0231 during the period of March 1, 1987 - Dec. 31, 1989. The project was motivated by the recognition that designing real-time distributed computer systems (DCS's) had been largely an artistic activity and a scientific foundation for reliable and systematic design existed only in a weak and incoherent state. Such foundation has become an important research issue in computer science and engineering due to the continuous increase in demands for ultra-reliable computer systems capable of supporting critical real-time applications. While the long term objective of this project was to contribute to the establishment of the scientific foundation for designing fault-tolerant DCS's with response time guarantee, more specific goals of the project were the following.

(1) Establish a real-time distributed computation model yielding simple techniques for response time guarantee,
(2) Develop DCS architectures possessing effective fault-tolerance capability, expandability, and high predictability of worst-case performance,
(3) Develop design environments supporting specification and validation of real-time behavior.

The research reported here constituted the second phase of a two-phase project. The preceding phase, Phase I, was conducted at the University of South Florida (USF) during the period of June 1, 1986 - June 30, 1987 (Contract No. N000014-86-K-0392-P00001). Therefore, a brief summary of the Phase-I research results is included in Section 3 before the results of the reporting phase (II) are summarized in Section 4.

2. Research Directions

At the early stage of this project, some design philosophies and study strategies which might distinguish this project from others were adopted. They can be summarized as follows.

(1) Make the processing deadlines explicitly treated attributes of both atomic and compound computation units.

(2) Distinguish formally between the real-time database and the archival database.

(3) Pursue deterministic time behavior in designing communication protocols, operating system (OS) structures, and application software.

(4) Build the distributed clock synchronization logic into a VLSI component in a network interface unit to achieve the microsecond level synchronization.

(5) Explore the fault tolerance (FT) schemes that can handle in a uniform manner both hardware faults and software faults, the latter including OS faults and application software faults.

(6) Identify the generic forward recovery techniques applicable to hard-real-time applications as clearly distinguished from others.

(7) Reflect unique characteristics of tightly coupled networks (TCN's) (e.g., a radar-tracking parallel computer system located at a single ground site) and loosely coupled networks (LCN's) (e.g., local area networks (LAN's) in factories or wide area networks in defense applications) in developing fault tolerance schemes and OS structures.

(8) Develop first the real-time FT schemes that can enhance the robustness of a computing station (a processing node executing a single application process) and then the supplementary schemes for making a group of cooperating computing stations fault-tolerant.

(9) Validate the formulated architectures, OS structures, and FT schemes not only via analyses and logical reasoning/proofs but also via experimental incorporation into real-time computer network testbeds built on TCN hardware, LCN hardware, and functional real-time application models.

3. Summary of the Phase-I Research Conducted at USF

Main results obtained during this phase are as follows.

3.1 A preliminary approach to specification of timing constraints during distributed computing system design and their validation

It was the premise of this research that the designs of real-time computer systems needed in critical applications must be rigorously verified for their capabilities for meeting the specified deadlines. Such designing with response time guarantee, called safe design here, is dependent upon the system configuration and the scheduling strategy used among other factors.

Specification of timing constraints is the very first step in safe design of real-time systems. One of the timing specification approaches studied may be called the time-tagged block approach. The basic idea is to specify timing constraints in association with execution blocks. During the Phase I the notion of completeness of a set of timing specification primitives needed to support the time-tagged block approach, was formalized and then a complete and practical set of primitives was established [Yan86].

Validation of time specifications is basically to check the feasibility of meeting the specifications at run-time. A preliminary version of an overall methodology for such validation for the case of distributed computer systems was formulated [Yan86]. The methodology uses various analytic verification techniques in its several constituent steps. The techniques are generally of two types: one that is machine-independent, and the other that is machine-dependent. The machine-independent techniques detect the inconsistency in the specifications and the impossibility of meeting the specifications, when the undesirable properties persist regardless of machine configurations and operating strategies. The other techniques reflect the machine characteristics in determining the execution feasibility.

3.2 A scheme for coordinated execution of independently designed recoverable distributed processes

A scheme for facilitating efficient backward recovery in loosely coupled networks (LCN's) was developed [You88]. The scheme, called the PTC/LCN (programmer-transparent coordination / LCN) scheme, is meant to be a fully general approach to facilitating efficient backward recovery in LCN environments where the autonomy of each process is highly desired. It shares the same basic design philosophy with the original PTC scheme proposed earlier in [Kim78], but was formulated to fit LCN systems unlike the original PTC scheme better suited for centralized

systems. The scheme allows independent and uncoordinated design of error detection and recovery capabilities of distributed processes. It makes provision for properly coordinating such distributed processes at run-time for cooperative recovery without incurring a cyclic chain of rollback propagations called a domino effect. The operational rules of the scheme were devised such that a minimal number of recovery-points (RP's) were used for maintaining the capability for recovery with minimum-distance rollbacks. These capabilities were formally proved.

The system design philosophy underlying the scheme is such that each process must be solely responsible for detecting the errors that it originates. An approach to making judicious exceptions, i.e., utilizing the cooperative error detection capabilities of processes without incurring a domino effect, was devised in order to further enhance the system robustness.

3.3 Testbed establishment and an evaluation of the DRB scheme

For rigorous validation of newly formulated design techniques and system structures, we adopted the approach of testbed-based validation. The availability of low-cost building-blocks such as microcomputers and interconnection devices, had made the construction of cost-effective DCS testbeds not much more expensive than constructing pure software simulators running on centralized computer systems. Testbeds are capable of representing the operating environment and input scenario more accurately than software simulators.

Initial versions of three major real-time distributed computing testbeds were established, each including a distributed real-time control program and a simulator of an application environment with sensor devices and actuators. The three testbeds deal with the three different types of real-time object tracking applications: (1) tracking with a ground-based radar, (2) tracking with a sensor boarded on a high speed moving vehicle, and (3) cooperative tracking by sensors distributed over multiple satellites. The first testbed was built around an in-house developed tightly coupled microcomputer network called the Macro Dataflow Network (MDN), the second testbed around another tightly coupled microcomputer network built by Unisys Corp., called the Crossbar Multi-microcomputer System (CMS), and the third testbed around a local area network manufactured by Cromemco Inc.

All the real-time distributed operating systems used in the three testbeds were developed in house. New techniques and tools were to be evaluated by integrating them into the testbed facilities and applying them to the experimental development of a practical network application system.

One of the design techniques evaluated by use of the MDN testbed is the distributed recovery block (DRB) scheme initially formulated in [Kim84]. The DRB scheme is based on a combination of both distributed concurrent processing and recovery block structuring concepts to achieve fast forward error recovery and to treat both hardware and software faults in a uniform manner with minimal execution overhead. It is an active redundancy scheme where multiple processors concurrently execute multiple versions of a software component and then the same acceptance test. The acceptance test in each processor, together with a watch-dog timer, checks reasonableness of the computational results of the version executed as well as the timeliness of the execution. The scheme was incorporated into the MDN testbed and subsequent measurement and evaluation demonstrated the fast recovery capability of the scheme and the soundness of the implementation strategies adopted [Yoo88].

The testbed facilities were transferred to UCI in early 1987 and have since been upgraded in major ways.

4. Results of the Phase-II Research Conducted at UCI

Main results obtained during the Phase II (the reporting period) are as follows.

4.1 Identification of critical research issues and promising research directions in real-time fault-tolerant distributed computing

An assessment was made of the state of the art in designing fault tolerance capabilities into real-time DCS's [Kim88c,Kim89d]. An emphasis was given to the problem of achieving system-level fault tolerance (i.e., an integration of hardware and software fault tolerance) in challenging real-time applications. Some of the guidelines considered to be highly useful in searching for schemes for system-level fault tolerance were established. One of the guidelines established was to distinguish between the real-time fault tolerance schemes that can enhance the robustness of a computing station (a processing node executing a single application process) and the supplementary schemes for making a group of cooperating computing stations fault-tolerant. Some of the established or promising approaches for "hardening" individual computing stations were assessed first and thereafter some of the promising approaches for hardening computing station groups were assessed. As a consequence of this state of the art assessment some critical research issues such as handling of interaction faults, validation of the software fault tolerance capability, etc., were identified.

These results were published in [Kim89c,Kim89d] copies of which are attached as Appendix A.I and A.II.

4.2 A Skeleton of the Foundation for Realizing System-Level Fault Tolerance

An initial framework of the scientific foundation for realizing system-level fault tolerance in real-time DCS's was established. The framework includes among others the DRB (distributed recovery block) scheme, the DCONV (distributed conversation) scheme, the PTC (programmer-transparent coordination) scheme, a TB (temporary blackout) handling scheme, and the complementary relationship among the schemes. These schemes enable the computer system to detect and recover from both hardware and software faults without missing the deadlines for processing important data and delivering outputs to the controlled object/environment.

4.2.1 Distributed recovery block (DRB) scheme

As mentioned before, the distributed recovery block (DRB) scheme can be used to obtain highly reliable computing stations (subsystems of DCS's) each dedicated to execution of a specific

real-time (atomic) task and capable of forward recovery from both hardware and software faults. During the reporting period, research on the DRB scheme was advanced in several areas. First, various issues that arise in implementing the DRB scheme were identified together with some promising approaches [Kim88a]. The issues in extending the DRB scheme with the capability of reincorporating a repaired node without disrupting the real-time computing service were also identified. These results were published in [Kim88a] a copy of which is attached as Appendix A.III.

Second, the operating principle of the DRB scheme was refined in the area of using watchdog timers in each node [Kim89a]. As a result, a logical basis was established for systematic use of watchdog timers in ensuring timely actions of both the partner node as well as the host node itself. A possible combination of the DRB scheme and a load balancing scheme suitable for implementation in shared memory multi-computer systems was evaluated with encouraging results in [Kim89a]. A copy of [Kim89a] is attached as Appendix A.IV.

Third, the concept of combining the DRB scheme with the pair-of-comparing-pair scheme used in systems such as Stratus for high-coverage detection of hardware faults and quick recovery was formulated [Kim88c,Kim89d]. In this DRB extension, each of the primary and backup nodes is actually a comparing pair of processors so that coverage of hardware faults may be significantly enhanced.

Finally, considerable amount of efforts were invested into experimental evaluation of the scheme. An implementation model of the DRB scheme extended with the capability of non-disruptive rejoin of a repaired node was incorporated into the MDN testbed [Kim88a]. The performance of the implemented system was then measured both during the normal fault-free processing and during the period experiencing faults. The experiment produced not only another evidence of the efficient forward recovery capability of the scheme but also validated the implementation model. In the MDN testbed, the DRB scheme was applied to only one computing station. In order to obtain an additional level of confidence in the soundness of the newly formulated implementation approaches as well as to obtain further insights into the performance impacts of the DRB scheme in real-time applications, an experiment that involved application of the DRB scheme to two adjacent computing stations in the CMS testbed was conducted [Min91]. The results again demonstrated the fast forward recovery capability of the DRB scheme as well as the effectiveness of the implementation approaches formulated.

Industry started taking the DRB scheme with serious interest. A small company located in Los Angeles, SoHaR, Inc., extended and thoroughly validated the DRB scheme for use in nuclear

reactor control applications under a SBIR (Small Business Innovation Research) grant from the US Department of Energy in recent years [Hec89,Hec91].

4.2.2 The conversation scheme and the distributed conversation (DCONV) scheme

A fundamental problem in designing error detection and recovery capabilities into cooperating asynchronous processes, arises from the possibility of erroneous process interaction [Ran75]. A promising approach to design of fault-tolerant interacting processes is the conversation scheme originally proposed by Randell in an abstract form [Ran75]. Based on the abstract model, we have derived some practical versions [Kim82,Kim85]. The conversation scheme is based on tightly coordinated design of error detection and recovery capabilities of interacting processes. It can be viewed as a two-dimensional (process & time dimensions) extension of the recovery block scheme.

During the reporting period, research on the conversation scheme was advanced in several different areas. First, fundamental performance characteristics of the conversation scheme were evaluated by use of an analytic model [Kim88d,Kim89c]. The results considered important here include the performance improvement achievable by exploiting the lookahead execution principle in reducing the overhead of synchronizing processes participating in conversations. These results were published in [Kim88d,Kim89c] and a copy of [Kim89c] is attached as Appendix A.V. Secondly, various approaches to implementing the conversation scheme in different types of LAN's were identified and their cost-performance study was performed [Yan89]. Important implementation factors considered include the control of exits of processes upon completion of their conversation tasks and the approach to execution of the conversation acceptance test. Two different exit control strategies, one in a synchronous manner and the other in an asynchronous manner, and three different approaches to execution of the conversation acceptance test, centralized, decentralized, and semi-centralized, were examined and compared in terms of system performance and implementation cost. A new efficient approach to run-time management of recovery information based on an extension of the recovery cache scheme was also discussed. The effectiveness of these execution approaches also depends on the way conversations are structured initially by program designers. Therefore, the two major types of conversation structures, Name-Linked Recovery Block (NLRB) and Abstract Data Type (ADT) Conversations, were examined to analyze which execution approaches are the most efficient for each conversation structure. These results provide useful guidelines for implementing conversations in loosely coupled DCS's. These results were published in [Yan89,Yan91] and copies are attached as Appendix A.VI and A.VII.

Finally, the conversation scheme was extended to incorporate the principle of concurrent execution of redundant software components that was exploited in the DRB scheme. The resulting scheme was called the distributed conversation (DCONV) scheme [Kim89d]. The DCONV scheme can be viewed as a fundamental approach to hardening a group of interacting computing stations in real-time TCN's or LAN's. The scheme is capable of achieving forward recovery when a part or all of a group of computing stations fail. Therefore, the DCONV scheme is an approach supplementary to the DRB scheme. The DRB scheme is applicable to non-interacting segments (i.e., atomic tasks) of application processes. To put it another way, it is a scheme to prevent a fault from crossing the boundaries between real-time processes as much as possible. For protecting against faults leaking through the guards established by the DRB scheme, supplementary schemes such as the DCONV scheme are needed. The DCONV scheme supports the coordinated design of interacting real-time distributed processes (strings of tasks) that are capable of tolerating faults arising in their interaction.

4.2.3 The programmer-transparent coordination (PTC) scheme

The PTC scheme is an approach to design of fault-tolerant interacting processes that is fundamentally different from the conversation scheme. The PTC scheme allows the design of error detection and recovery capabilities of a process to be made in a manner independent of recovery structures of other cooperating processes. Also, a recoverable region of each process may involve any finite sequence of message exchanges. The PTC scheme is attractive in applications where it is desirable to build some autonomy into each of the distributed nodes due to relatively expensive and unreliable inter-node communication, security concerns, etc.

In regards to the run-time aspect, the essence of the PTC scheme is to facilitate run-time coordination of independently designed processes for their cooperative error detection and recovery with the aid of an intelligent system kernel. The intelligent kernel must be capable of supporting the processes in such a way that the so-called domino effect [Ran75], which refers to an intolerably long chain of process rollbacks which can occur when the recovery points of interacting processors selected by the programmer are not well coordinated and the conventional kernel is used, does not occur. Such a kernel must be capable of automatically establishing appropriate recovery points of interacting processes including those not specified by the programmer.

During the reporting period, research on the PTC scheme was advanced in several areas. First, the protocols for cooperation among distributed intelligent kernels that are optimal in the sense that they incur minimal overhead in both state saving and recovery, have been formulated in

a highly general form [Kim88b]. To maximize the generality of the protocols, a model of a DCS in which distributed computing nodes communicate via shared data structures (rather than message passing) was adopted as the operating environment. A copy of [Kim88b] containing these results is attached as Appendix A.VIII.

Secondly, efforts were made to enhance the capability of the scheme for real-time recovery. As a result, an extension of the PTC scheme where a higher degree of autonomy is allowed to each process than in the original scheme was formulated [Kim88c] and studied further in [Kim89d]. This extension, called the PTC/AR (PTC with adaptive receivers) scheme, appears to be an opening of a new meaningful research direction since unlike many proposed approaches to design of adaptive fault-tolerant systems, it offers a systematic way based on the PTC logic for realizing autonomous survival capabilities in real-time computer networks. The PTC/AR scheme provides a framework in which both backward and forward recovery capabilities can be adaptively employed. Development of system aids supporting the use of the PTC/AR scheme is an important topic for future research.

4.2.4 Temporary blackout (TB) handling

The temporary blackout (TB) that disrupts orderly operation of electronic components and erases the contents of registers and RAM's, occurs in many applications due to unreliable power sources, high energy events, etc. In a sense, the TB handling deals with global faults of a distributed system. It is indispensable in many aerospace applications where such global faults are non-negligible. In real-time distributed systems, processes must establish their recovery points in a coordinated manner so that the system can restore itself to a consistent state. In order to identify the design issues in providing TB handling capabilities in DCS's, an experimental design of the TB handling capability into a real-time DCS testbed was conducted. Promising design approaches including those for structuring state-saving actions and cooperative recovery actions of distributed processes were formulated [Kim88e,Kim89b]. A main research issue being addressed here is how far the checkpointing part can be made to be programmer-transparent (i.e., done without burdening the program designer) while the real-time forward recovery capability is not compromised. Here the forward recovery logic is the responsibility of the program designer. Copies of [Kim88e] and [Kim89b] are attached as Appendix A.IX and A.X, respectively.

4.3 Model of real-time distributed computation

The goal of this research task is to develop a computational model and design methodology that will greatly simplify the problem of designing a DCS with a guarantee that the

DCS will not miss deadlines under peak load conditions. (This research is a collaborative effort with Hermann Kopetz in Austria who is also a visiting scholar at UCI.)

The very first step in the design of real-time systems with guaranteed response should be the specification of timing constraints. In the course of this specification study, we came to realize the need for a computation model which embodies not only the concept of explicit timing specification but also the notion of the real-time database that is clearly distinguished from that of the archival database. The first abstract version was published [Kop89] and its copy is attached as Appendix A.XI.

Subsequent work led to the formulation of the notion of real-time objects [Kop90] which are distinguished from the conventional object model in three major ways:
(1) For each action of an RT-object a deadline is imposed;
(2) For some actions of an RT-object a real-time clock serves as the mechanism for triggering the object actions as a function of real time; and
(3) Real-time data contained in an RT-object become invalid after the interval called the maximum validity duration passes.
The work in this area is continuing toward the goal of completing the formation of an object-based model of real-time distributed computation and subsequent development of temporal behavior specification and validation techniques.

4.4 Efficient diagnosis of hypercube multicomputer systems

As a step toward development of efficient diagnosis strategies for real-time DCS's, efforts were made to establish some theoretical basis for efficient diagnosis of hypercube multicomputer systems. As a result, an approach in which processors test each other and the test results are collected and analyzed to determine faulty processors, was formulated for diagnosis of hypercubes [Kav91]. While this approach is an extension of the approach formulated by Preparata and others [Pre67], this new approach has significantly greater diagnostic power and is quite different from the approaches studied by others for hypercubes. First, the new approach is a pessimistic diagnosis strategy in that sometimes a good processor surrounded by faulty processors is removed for repair as well. This is in contrast to the precise strategy in which only faulty processors are removed for repair. The new pessimistic approach raises the degree of diagnosability of a hypercube considerably beyond the degree achieved under the precise diagnosis strategy. In fact, we proved that the degree of an n-cube is 2n-2 under the pessimistic diagnosis strategy and n under the precise diagnosis strategy. Moreover, to achieve the diagnosability of degree n under the pessimistic strategy one needs to use only $\lceil n/2 \rceil + 1$ interprocessor links per processor as

testing links. Therefore, this approach enables to some extent concurrent diagnosis, i.e., execution of the diagnosis procedure while some application computations are in progress. In addition, an algorithm for selecting $(\lceil n/2 \rceil + 1)*n/2$ bidirectional links in an n-cube for use as testing links was developed. A copy of [Kav91] containing these results is attached as Appendix A.XII.

4.5 Testbed establishment

In order to support various experimental studies mentioned in Section 4.2, three of the real-time computer network testbeds established in the UCI DREAM (Distributed Real-time Ever Available Microcomputing) Laboratory had to be upgraded. Various design issues encountered in the course of establishing two of the basic microcomputer network testbed facilities and augmenting them to support some fault tolerance experiments conducted were summarized in [Kim89b]. The main emphasis is on the operating system capabilities identified as necessary to support the fault tolerance schemes experimented with. Further details on the three testbeds as well as a description of the recently added fourth testbed are contained in Appendix A.XIII.

5. Conclusion

The results summarized in this report represent some advances in the state of the art in the design of fault-tolerant real-time DCS's as well as substantial addition to the knowledge base related to fault-tolerant real-time distributed computing. Although potential benefits of the formulated approaches seem substantial, some of them still remain in immature states. Much further research is needed to convert the appealing concepts into convincing demonstration of the benefits . The following study topics are among those considered to be highly worthwhile subjects for research in the near future.

(1) Development of a complete object-based model of real-time distributed computer systems,

(2) Development of techniques for temporal behavior specification based on the object-based system model,

(3) Development of efficient techniques for dynamic allocation of resources to meet temporal behavior requirements,

(4) Development of efficient techniques for temporal behavior validation,

(5) An integration of the DRB scheme and a practical network diagnosis scheme,

(6) An experimental study of the conversation scheme and the DCONV scheme,

(7) An experimental study of the PTC scheme.

6. References

6.1 Publications from the Phase-I research conducted at USF

[Yan86] Yang, S.M., 'Timing Specification and Verification for Fault-Tolerant Distributed Computer Systems', Ph.D. Dissertation, Dept. of Computer Sci. & Engr., Univ. of South Florida, Dec. 1986.

[Yoo88] Yoon, J.C., 'An Approach to Design of Fault-Tolerant Real-Time Tightly Coupled Networks and Its Experimental Validation', Ph.D. Dissertation, Dept. of Computer Sci. & Engr., Univ. of South Florida, May 1988.

[You88] You, J.H., 'Techniques for Efficient Implementation of Fault-Tolerant Loosely Coupled Networks Based on Decentralized Computation Recovery', Ph.D. Dissertation, Dept. of Computer Sci. & Engr., Univ. of South Florida, May 1988.

6.2 Publications from the Phase-I research conducted at UCI

[Kav91] Kavianpour, A. and Kim, K.H., "Diagnosabilities of Hypercubes under the Pessimistic One-Step Diagnosis Strategy", IEEE Transactions on Computers, Vol.40, No.2, Feb. 1991, pp.232-237.

[Kim88a] Kim, K.H. and Yoon, J.C., "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme", Proc. IEEE Computer Society's 18th Int'l Symp. on Fault-Tolerant Computing, June 1988, pp.50-55.

[Kim88b] Kim, K.H., "Programmer-Transparent Coordination of Recovering Parallel Processes: Philosophy and Rules for Efficient Implementation" IEEE Trans. on Software Engineering, Vol.14, No.6, June 1988, pp.810-821.

[Kim88c] Kim, K.H., "Designing Fault Tolerance Capabilities into Real-Time Distributed Computer Systems", Proc. IEEE Computer Society's Workshop on the Future Trends of Distributed Computing Systems in the 1990's, Hong Kong, Sep. 1988, pp.318-328.

[Kim88d] Kim, K.H. and Yang, S.M., "An Analysis of the Performance Impacts of Lookahead Execution in the Conversation Scheme", Proc. IEEE Computer Society's 7th Symp. on Reliable Distributed Systems, Columbus, Oct. 1988, pp.71-81.

[Kim88e] Kim, K.H., "Issues in Design of Temporary Blackout Handling Capabilities into Tightly Coupled Computer Networks", Proc. 2nd Int'l Software for Strategic Systems Conf., Huntsville, AL, Oct. 1988, pp.92-99.

[Kim88f] Kim, K.H., "Design of Fault-Tolerant Distributed Computer Systems with Response Time Guarantee", Proc. ONR Foundations of Real-Time Computing Research Initiative Kickoff Workshop, Falls Church, VA, Nov. 1988, pp.101-106.

[Kim89a] Kim, K.H. and Welch, H.O., "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults in Real-Time Applications", IEEE Trans. on Computers, May 1989, pp.626-636.

[Kim89b] Kim, K.H., "An Approach to Experimental Evaluation of Fault-Tolerant Distributed Computing Schemes", IEEE Transactions on Software Engineering, June 1989, pp.715-725.

[Kim89c] Kim, K.H. and Yang, S.M., "Performance Impacts of Look-Ahead Execution in the Conversation Scheme", IEEE Transactions on Computers, August 1989, pp.1188-1202.

[Kim89d] Kim, K.H., "Approaches to System-Level Fault Tolerance in Distributed Real-Time Computer Systems", Proc. 4th Int'l Conf. on Fault-Tolerant Computing Systems, Baden-Baden, W. Germany, Sept. 1989, pp.268-281 (published in the Lecture Notes series by Springer-Verlag) (Invited paper).

[Kop88] Kopetz, H. and Kim, K.H., "Consistency Constraints in Distributed Real Time Systems", in M.G. Rodd and T.L. d'Epinay eds., 'Distributed Computer Control Systems 1988', Pergamon Press, 1988 (Proc. 8th IFAC Workshop on Distributed Computer Control Systems, Vitznau, Swiss, Sep. 1988), pp.29-34.

[Min91] Min, B.J., "Efficient Implementation of the Distributed Recovery Block (DRB) Scheme in Highly Parallel Computer Systems", Ph.D. Dissertation, Dept. of Electrical & Computer Engineering, Univ. of Calif., Irvine, Aug. 1991.

[Yan89] Yang, S.M. and Kim, K.H., "Implementation of the Conversation Scheme in Loosely Coupled Distributed Computer Systems", Proc. IEEE Computer Society's 9th Int'l Conf. on Distributed Computing Systems, June 1989, pp.570-578.

[Yan91] Yang, S.M. and Kim, K.H., "Implementation of the Conversation Scheme in Message-Based Distributed Computer Systems", Tech. Rept. UCI-ECE-90-12, Dept. of Electrical & Computer Engineering, UCI, October, 1990. To appear in IEEE Transactions on Parallel and Distributed Systems.

6.3 General references

[Hec89] Hecht, M., Agron, J., and Hochhauser, S, "A Distributed Fault Tolerant Architecture for Nuclear Reactor Control and Safety Functions", Proc. IEEE Computer Society's 1989 Real-Time Systems Symp., Dec. 1989, pp.214-221.

[Hec91] Hecht, M., Agron, J., Hecht, M., and Kim, K.H., "A Distributed Fault Tolerant Architecture for Nuclear Reactor and Other Critical Process Control Applications", Proc. IEEE Computer Society's 21st Int'l Symp. on Fault-Tolerant Computing, June 1991, Montreal, pp.462-469.

[Kim78] Kim, K.H., "An Approach to Programmer-Transparent Coordination of Recovering Parallel Processes and Its Efficient Implementation Rules", Proc. 1978 Int'l Conf. on Parallel Processing, August 1978, pp.58-68.

[Kim84] Kim, K.H., "Distributed Execution of Recovery Blocks: an Approach to Uniform Treatment of Hardware and Software Faults", Proc. 4th Int'l Conf. on Distributed Computing System, May 1984, pp.526-532.

[Kim86] Kim, K.H., You, J.H., and Abouelnaga, A., "A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes", Proc. IEEE Computer Society's 16th Int'l Symp. on Fault-Tolerant Computing, July 1986, pp.130-135.

[Kop90] Kopetz, H. and Kim, K.H., "Temporal Uncertainties in Interaction among Real-Time Objects", Proc. IEEE Computer Society's 9th Symp. on Reliable Distributed Systems, Huntsville, AL, Oct. 1990, pp.165-174.

[Pre67] F. P. Preparata, G. Metze and R. T. Chien, "On the Connection Assignment Problem of diagnosable Systems," IEEE Trans. on Elec. Computers, vol. EC-16, Dec. 1967, pp.848-854.

[Ran75] Randell, B., "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, June 1975, pp.220-232.

Appendix A.I

Designing Fault Tolerance Capabilities into Real-Time Distributed Computer Systems

Designing Fault Tolerance Capabilities into Real-Time Distributed Computer Systems

K.H. Kim

Computer Engineering Program, Dept. of Electrical Engineering
University of California Irvine, California 92717

Abstract

The purpose of this paper is to provide a brief assessment of the state of the art in designing fault tolerance capabilities into real-time distributed computer systems (DCS's) and to summarize major issues that remain unresolved in this field. The paper starts with a discussion of some of the important characteristics of real-time DCS's that must be carefully reflected in designing fault tolerance capabilities. Some fault tolerance schemes that have been identified as promising ones for use in real-time DCS's are reviewed and then major remaining issues are discussed. The issues discussed include handling of both interaction faults and software faults, evaluation of the overall effectiveness of a combination of the fault tolerance schemes, etc.

1. Introduction

The steady increase observed during the past decade in use of distributed computer systems (DCS's) in safety-critical real-time applications is expected to continue through 1990's. For example, DCS's have been increasingly adopted in applications such as space navigation, air-traffic control, hospital automation, national defense, etc. [Bha87,Dav80,Fou84,Hec76,McD82, Ram81]. In order to attain the desired level of reliability, such DCS's must be designed to possess effective fault tolerance capabilities.

On the other hand, missing deadlines on the part of the DCS's is quite often as dangerous as producing incorrect values. Therefore, if incorporation of fault tolerance mechanisms increases the chance of a DCS missing deadlines, it would rather decrease the overall system reliability than improving it. A DCS used in an application where the controlled objects cannot tolerate late arrival or omission of even a single control output of the controlling computer system, is called a hard-real-time DCS.

Designing hard-real-time DCS's is a relatively immature branch of computer engineering. Designing fault tolerance capabilities into hard-real-time DCS's is even less understood. The purpose of this paper is to discuss major issues that need to be resolved in 1990's in order to make design of such fault tolerance capabilities a more widely practicable engineering process

The paper starts in section 2 with a discussion of some of the important characteristics of real-time DCS's with respect to provision of fault tolerance capabilities. Section 3 then provides a review of some of the fault tolerance schemes that have been identified as promising ones for use in real-time DCS's. Potential strengths as well as limitations of the schemes are discussed. Major issues that remain to be resolved in 1990's are discussed in section 4. The issues considered important here include handling of both interaction faults and software faults, provision of autonomy and adaptiveness into the nodes handling certain unforeseen faults, validation of real-time recovery, etc. Section 5 is the summary section.

2. Characteristics of Real-Time DCS's

2.1 Important characteristics

In designing fault tolerance capabilities into real-time DCS's, the following system characteristics must be carefully reflected. First, the processing nodes exchange information among themselves. This interaction may cause the fault propagation through the network. That is, faulty behavior of one node may cause the failure of another node(s). Therefore, cooperation among nodes for fault detection and recovery is needed [Ran75,Kim82] Secondly, timely recovery is another factor to be considered in designing a fault-tolerant real-time DCS In a real-time environment (especially in the hard-real-time environment), outputs of the computer system should be correct not only in logic but also in time [And83,Hec76,Kop85] Finally, the network structure of a system has a significant effect upon the design of the operating system as well as fault tolerance mechanisms incorporated in it [And75]. In this paper, DCS's are classified into two types, i.e., tightly coupled network (TCN) and loosely coupled network (LCN)

TH0228-7/88/0000/0318$01.00 © 1988 IEEE

2.2 TCN versus LCN

There are several parameters which can be used to classify DCS's into TCN's and LCN's. One way of classifying DCS's is based on the geographical dispersion of the system, i.e., maximum distance between two processing nodes in the system. With this classification criterion, a system whose processing nodes are located in a single room can be classified into a TCN. LCN's are further divided into local area networks (LAN's) and wide area networks (WAN's). A LAN spans a single building or several buildings, such as those on a college campus. This classification is more often used in classifying LCN's into LAN's and WAN's than classifying DCS's into TCN's and LCN's.

Another way of classifying DCS's is based on the data transmission speed, to be more precise, byte transmission time and message transmission time, between two processing nodes in the system. In TCN's, the byte transmission time should be close to the time for accessing local memory in the node. In addition, message transmission should be accomplished with minimum protocol overhead. Although this classification scheme seems logically sound, it is difficult to select a pair of fixed numbers for the byte and message transmission times that can be used as boundaries between TCN's and LCN's. The difficulty largely stems from the wide discrepancy in transmission speed among different transmission media, e.g., twisted pair, optical bus, various types of shared memories, etc. In some sense, this difficulty is natural because the term "tight" and "loose" are relative terms.

The third way of classifying DCS's is based on the communication medium adopted in the system. A DCS which communicates through a shared (or common) memory or a high-speed parallel bus can be classified as a TCN. In the case where a high-speed parallel bus is used, each processing node is equipped with an IO processor/controller dedicated to communication with other nodes. In some sense, this classification scheme is a simplistic adaptation of the aforementioned logically sound scheme based on the byte and message transmission times. In the rest of this paper, we primarily use this classification scheme because of its simplicity. Some examples of the TCN and the LCN are given below.

An example of a TCN with shared memory is the CMS (Crossbar Multi-Microcomputer System) built by Unysis Corp. [Mcd82,Chu87]. In this system, microcomputers access shared memory modules through a crossbar connection network. An example of a TCN with high-speed parallel busses can be found in the Tandem system architecture [Kat78b]. Nodes are connected through a high-speed parallel bus, named Dynabus, and each node is equipped with an interprocessor bus controller. Performance of the communication subsystem in this structure is comparable to that observed in TCN's with shared memories, provided that the number of busses is the same in both structures.

LCN's are largely divided into LAN's and WAN's. Classification parameter in this case is the geographical distance between the nodes in the system. Examples of the LAN architecture are abundant in the literature, e.g., [Sta84,Iha84].

WAN's in general are designed to facilitate data communication among geographically dispersed sites. Each site of a WAN can be a terminal, a computer system, a TCN, or a LAN. For example, the Pluribus multiprocessor system which is a TCN, serves as an interface message processor (IMP), i.e., a node in the ARPA network [Kat78a]. The StrataNET connects a set of Stratus systems which are TCN's [Str84]. In WAN's, leased lines, packet satellite, and packet radio are the most commonly used communication media.

The significant gap that exists between the inter-node communication costs in real-time LAN's and those in real-time WAN's has an important impact on the fault tolerance and other system structuring areas. For example, the autonomous survival of each node in a real-time WAN seems to be a highly desirable capability, whereas importance of making such provision is much less in real-time LAN's. In addition, there can arise a question as to whether the designer of software objects or modules needs to be concerned with the locations of the objects. Our view on location transparency is as follows. While provision of location transparency simplifies the job of an object designer, it results in increased inter-node communication cost. This is because knowledge in inter-object communication patterns cannot be easily exploited in placement of objects. Therefore, it may often be easily justified in real-time LAN's whereas it is not much useful in real-time WAN's where communication costs are intrinsically high.

3 Promising Fault Tolerance Schemes Established

This section (3) is by no means intended to be a thorough review of the fault tolerance schemes applicable to real-time DCS's. Just a few schemes that are of fundamental nature and also indicative of the state of the art in real-time fault-tolerant distributed computing are briefly reviewed. Before those selected schemes are reviewed, the fault model adopted is discussed in section 3.1, various primitive steps involved in effecting fault tolerance in section 3.2, and the desirable types of fault tolerance schemes in section 3.3.

3.1 The types of faults considered

In order to make the remainder of this paper relevant to a broad range of challenging real-time applications, the following general fault model is adopted in this paper.

(1) In many challenging real-time application, it is infeasible to completely avoid software faults. Therefore, both hardware faults and software faults are considered.

(2) The types of hardware faults considered include both permanent faults and transient faults. Components such as CPUs, buses, memories, peripherals, and communication links are regarded as subject to possible failures. Also, common-mode failures such as temporary power outages are non-negligible types of faults in some environments.

(3) The types of software faults considered include both algorithmic faults and timing faults. As regards the possible locations/sources of the faults, both application software and system software such as operating system are considered equally. No assumption is made about the existence of any kind of perfect firewalls for fault containment.

3.2 An abstract model of fault-tolerant DCS structure.

Various primitive steps involved in effecting fault tolerance are now discussed. Tolerance of component failures during system operation is generally realized through the four steps depicted in Figure 1. Once a system enters a faulty state, the fault must first be detected. Secondly, the source of the fault must be isolated at the level of replaceable/repairable components. The fault source may be hardware faults and/or imperfect software. Since software runs on hardware, it is natural to first test hardware to see if it is operational. If any malfunctioning hardware modules are detected, appropriate hardware reconfiguration (repair) actions, e.g., replacement of the detected malfunctioning modules with spare modules, are taken. The third step is to restore the system to a state where the system can resume the application-oriented computation which was interrupted due to the fault. This often involves reloading the memory with the previously saved state information of the computation (i.e., contents of variables). After this computation state recovery is completed, some suspected software modules may be replaced with their alternate versions if available.

A generalization of the model to reflect the common characteristics of DCS's is the model depicted in Figure 2. There are largely four layers of components in the system structure. The bottom layer is a hardware-software layer and the other three are software layers. Multiple hardware nodes together with an interconnection or communication network form the hardware part of the bottom layer. Each node contains a software nucleus which supports coexistence of software components belong to the upper three layers within the node and handles diagnosis and reconfiguration of the node hardware.

The software components in the second layer which are distributed among multiple nodes are responsible for diagnosing the health of the system hardware and if necessary, reconfiguring it to establish an operational configuration. Suppose the software nucleus in each node (to be exact, the node diagnosis and reconfiguration (NDR) component) either fails to establish an operational state of the node or makes an erroneous report on the health of the node. The software components in the second layer can correctly diagnose the node status by using other nodes and if necessary, arrange for functional or physical replacement of a bad node by other nodes. The interconnection networks are also checked out by the second layer.

The software components in the third layer are responsible for establishing a computation state from which the execution of the application processes in the top layer can start or resume. These software components are also distributed among multiple nodes and may accomplish state establishment in a completely decentralized fashion or under some centralized control.

The software components in the top layer are distributed application processes communicating among themselves. The software components that are actively running on nodes most of the time are the application processes and the software components in the second and third layers are invoked upon detection of faults and also for periodic diagnosis.

In the model fault detectors are embedded in all four layers including the node hardware in the bottom layer. Once a fault is detected, the followup action sequence starts from the bottom layer regardless of the location of the component that detected the error. That is, the set of nodes to be involved in the followup actions is identified on the basis of the information supplied on the basis of the information supplied by the fault detector, and the software nucleus of each selected node diagnoses the node and attempts to establish an operational state of the node. Often all the nodes in the system are involved because the fault detector does not provide any clue on the damaged part of the system. The second layer is then called in to establish an operational system hardware configuration. The third layer is called in next to establish a consistent state of distributed application processes. This often involves reloading the memory with the previously saved state information of the processes (i.e., contents of variables). If neither the software nucleus of each node nor the software components in the second layer have detected any hardware malfunctioning, then the detected fault could have been caused by transient hardware fault or a residual error in the application software. Therefore, after the computation state recovery is completed, some suspected modules in the application processes may be replaced with their alternate versions if available.

3.3 Desirable types of fault tolerance schemes.

In view of the characteristics of real-time DCS's

discussed in Section 2 and the fault model adopted in Section 3.1, the following types of fault tolerance schemes are considered to be most desirable.

(1) The schemes that can cover both hardware faults and software faults: This type of schemes are highly desired, considering the potential complexity of large DCS's needed in some real-time applications.

(2) The schemes that can work under stringent time constraints: In hard-real-time environments, the time is the most precious resource and fault tolerance schemes with significant run-time overhead are at the least useless, if not harmful.

In addition, the schemes that facilitate cooperative detection of and recovery from faults are desirable in LCN's. Using this type of schemes is an essential requirement in realizing fault tolerance in LCN's because the modularity of LCN's can best be maintained and exploited for high degree of fault tolerance only by use of such schemes. Unfortunately, such schemes have not been well established.

The remainder of Section 3 will dwell mostly on the fault tolerance schemes that possess some of the characteristics mentioned above.

3.4 The comparing pair scheme and the scheme of pair of comparing pairs

The comparing pair scheme was adopted in No. 1A ESS and No. 3A systems [Toy78]. In this approach all critical components, such as processor and memory, are duplicated and outputs are compared for error detection. If a mismatch occurs, an interrupt is generated, which causes the fault-recognition program to run. The basic function of this program is to determine which half of the system is faulty. The suspected unit is removed from service and an appropriate diagnostic program is run to pinpoint the defective circuit pack.

The advantage of this scheme is the simplicity of the fault detection logic. Hardware reconfiguration and recovery are somewhat time-consuming because the fault-recognition program should be executed to find faulty unit. Like most hardware schemes, this mechanism is not applicable to handling residual software errors in the system. In addition to the comparing pair mechanism, special fault detection software, named audit programs, is also incorporated in ESS systems. Audit programs are called in to validate and check for consistency of data in the memory and peripheral equipment status.

An extension of the comparing pair mechanism, named pair of comparing pairs here, in which major functions are replicated four times, is used in Stratus system [Str84]. Each subsystem (a printed circuit board) has identical counterpart, its spare. Each subsystem is a comparing pair. A subsystem and its spare are tightly synchronized. Should one subsystem detect an internal mismatch, it stops the operation and its spare continues to carry the load.

The main advantage of this scheme is in that no special recovery logic is needed. Recovery time is almost negligible. This scheme also does not deal with software residual errors. The scheme requires a relatively large amount of hardware resources.

3.5 The triple modular redundancy (TMR) scheme

The essence of the TMR scheme is to use three versions/copies of a computing component and take a vote with their execution results. When there is discrepancy, the result with a majority vote is used.

Although the TMR scheme was initially developed as a scheme for hardware fault tolerance, it is in principle applicable to software fault tolerance as well, e.g., N-version programming scheme [Avi85]. However, its effectiveness when applied to complex program components is unknown at present. The voting approach requires design of multiple versions expected to generate truly identical computation results This could be a severe restriction in cases where complexity of a program component is high.

When the TMR scheme is used for hardware fault tolerance in DCS's, the critical issue is how to realize voting without degrading both the modularity of the system structure and the execution efficiency [Kat78a,Wen78]. Until this and other problems get resolved, the application domain of the TMR may be restricted to be within a single node in DCS's.

The TMR scheme can be applied to atomic computation steps, i.e., non-interacting segments of processes. It can also be applied to interacting segments of processes and object actions, provided that voting takes place before each interaction point of a process.

3.6 Checkpointing and temporary blackout handling

Checkpointing refers to saving the state of computation at various execution points called recovery points (RP's) so that when a fault occurs and the damage is contained, the computation may roll back to the most recent RP and restart from there. In the case of the Tandem system [Bar78], for each running process there is a corresponding identical process in another processor. This backup process replaces the primary process if an unrecoverable fault occurs in the primary's processor. The primary process sends to its backup periodic "checkpoint" messages, which define the states of the process at critical points in the computation. The operating system nucleus in each processor "wakes up" the relevant backup process upon discovering that its corresponding primary has failed The backup can then resume the task from the state defined in the last

checkpoint. In order to detect the failure of other nodes, each processor in the system sends a special "I'm alive" message every second to all processors in the system. Every two seconds, each processor checks to see that it has received one of these messages from each processor. If a message has not been received, then the former processor assumes that the latter processor is down.

One problem with this kind of backward recovery scheme based on checkpointing is the long recovery time taken. Consequently, its usefulness in real-time applications is somewhat limited. However, in some environments, temporary blackout events due to temporary power outages which affect a large subset of processing hardware and communication links, etc., cannot always be avoided. In such environments, a checkpointing-based backward recovery scheme is essential. There are two major aspects in this scheme. One is to dynamically establish recovery lines. (i.e., a coordinated set of RP's of interacting processes) as interacting processes progress while the other is to effect fast forward recovery upon expiration of the blackout. The latter part, i.e., fast forward recovery, basically involves reading the current status of the environment, comparing it against the state at the recovery line, and bringing the state of computation forward to an appropriate, not necessarily the best, state.

3.7 The DRB (distributed recovery block) scheme

The DRB scheme depicted in Figure 3 [Kim84,Kim88a] is essentially an active redundancy scheme where multiple processors concurrently execute multiple versions of a software component and then the same acceptance test. Each processor uses a time-out mechanism, as well. An important advantage of this scheme is that fast forward recovery can be achieved with the software redundancy produced through the aid of a convenient design tool, i.e., the recovery block [Hor74,Ran75]. In order to be more exact, the definition of recovery block is needed.

Recovery block consists of one or more routines, called try blocks here, designed to compute the same or similar results, and an acceptance test which is an expression of the criterion used for accepting the results of try blocks. For the sake of simplicity in description, a recovery block is assumed to contain only two try blocks, i.e., the primary and the alternate. The basic idea of the DRB scheme is to distribute the two try blocks to two different nodes and assign the acceptance test to both nodes. Both nodes use watchdog timers. After receiving the common input data from the predecessor computing station, each node executes a try block and then the acceptance test. If the node running the primary try block passes the test in time, it sends a signal "I'M OK" to the backup node (running the alternate try block). It then outputs the results to the successor computing station. As long as the primary node sends the signal to the backup node in time, the latter suppresses its own results. Therefore, only if the primary node dies or fails to pass the acceptance test, the backup node will start outputting its results.

This approach has two useful characteristics:

(a) Recovery can be accomplished in the same manner regardless of whether a node fails due to a hardware fault or a software fault, i.e., it is unnecessary to distinguish between hardware faults and software faults;

(b) The recovery time is minimal since maximum concurrency is exploited between the primary and the backup nodes.

The experimental study reported in [Kim84,Chu87] and subsequent studies have clearly demonstrated high-speed recovery capability of this scheme.

The DRB scheme is basically applicable to non-interacting segments of application processes and object actions.

3.8 Replication of files and objects

Delta system [Lel81] which uses a distributed executive for real-time transaction processing, supports replication of data objects. For example, three data objects, A,B, and C, may be replicated in three different data management (DM) processors, so as to ensure that they would be accessable even when one or more of the DM processors containing them had failed. The operating system supports maintaining the consistency of data objects. In addition, a checkpoint scheme has been implemented in order to ensure reliable execution of transactions. Similar mechanisms can be found in the RSEXEC system [For78].

Replicating more general types of objects, e.g., those defined in [Lis83,Tri83] and involving both state variables and atomic actions, is an approach applicable to a broader range of applications. An object replication scheme based on the DRB principle has been identified as a highly promising technique for use in real-time LAN applications. The scheme is depicted in Figure 4. In this scheme, the state variables are duplicated between the primary and backup objects. To each action/operation in the primary object, there corresponds an action/operation in the backup object. An important characteristic to note here is that the corresponding pair of actions/operations are not necessarily identical, although functionally similar, because they are alternate try blocks in a recovery block. A request for an object action is broadcasted to both the primary and backup objects. Generally both primary and backup object actions and subsequent acceptance tests are carried out concurrently. If both are successful, only the primary object delivers the responses to the client. The extent of exploiting concurrency between primary and

backup object actions as well as the commitment protocols differs among several cases of objects. In the worst case, the scheme degrades down to the well-known primary copy scheme for object replication. To be more specific, in the case of stand-alone (self-contained) objects that do not use other objects, the DRB-based scheme can be applied in its full functionality. In this case, the scheme provides the forward recovery capability. On the other hand, in the case of composite objects that use other objects by calling for either independent actions or nested actions of read-write type, the backup object may merely wait without executing its action until the primary object action is complete. If the primary object action is successful, the backup object will simply copy the result. Otherwise, the backup object will start its action. That is, the DRB-based scheme is reduced to the primary copy scheme in such a case.

3.9 The conversation scheme

The conversation is a conceptual extension of the recovery block which is a concrete language construct designed to support structuring the error detection and recovery actions of a single process. It is a two-dimensional enclosure of recoverable activities of multiple interacting processes, i.e., recoverable interacting session [Kim82,Ran75]. The enclosure consists of a recovery line, a test line, and two side walls defining exclusive membership as shown in Figure 5. A recovery line is a coordinated set of the recovery points (RP's) of interacting processes that are established (possibly at different times) before interaction begins. When all the processes roll back to the recovery line, there cannot be any further propagation of rollbacks among processes. A test line is a correlated set of the acceptance tests of the interacting processes. A conversation is successful only if all the interacting processes pass their acceptance tests forming the test line. If any of the acceptance tests fails, all the processes roll back to the recovery line and retry with their alternate try blocks. These alternate try blocks collectively define an alternate interacting session, whereas the set of primary try blocks executed first after the processes enter the conversation define the primary interacting session. Therefore, processes cooperate in error detection, regardless of the source of the error. Two sidewalls defined by a conversation imply that the processes participating in the conversation must neither obtain information from nor leak information to a process not participating in the conversation. That is, no "information smuggling" by processes in a conversation is permitted.

On the basis of the aforementioned abstract notion of conversation, practical language tools that support conversation design have been developed [Kim85].

One problem with the conversation scheme is the execution overhead incurred due to the synchronization imposed on the participants at the test line. An analytic evaluation has revealed that this overhead can be a very serious problem [Kim86a]. An approach to reducing the overhead is to permit the conversation participants finished early to proceed to exit from the conversation. An analytic evaluation has shown that such permission of asynchronous exits from conversations can lead to significantly reduced overhead [Kim88b].

Also, as described above, the conversation scheme facilitates backward recovery. However, the distributed execution principle exploited in the DRB scheme can also be incorporated into the conversation scheme to facilitate forward recovery.

The conversation scheme goes beyond the DRB scheme in that the former can handle interaction faults Both schemes are aimed at tight error containment by defining rigid enclosures for possibly faulty activities. The primary application domain of the conversation scheme is in real-time LAN environments. In spite of the recognized potential of the conversation scheme, a convincing demonstration of the utility of the scheme in real-time applications has yet to take place.

3.10 The PTC/LCN (programmer-transparent coordination in loosely coupled networks) scheme

In dealing with the domino effect problem [Ran75], one can conceive of two different approaches; the coordination-by-programmer approach and the coordination-by-machine approach. With the first approach, the program designer is fully responsible for designing processes such that they establish RP's in a well-coordinated manner. The conversation scheme is a representative example of this approach. On the other hand, the coordination-by-machine approach is aimed at relieving the program designer of the burden of coordinating the RP's of interacting processes Instead, the approach relies on an "intelligent" underlying processor system (or a virtual machine) which automatically establishes appropriate recovery points (RP's) of interacting processes.

Among a few coordination-by-machine schemes proposed in the literature, the programmer-transparent coordination (PTC) scheme developed in [Kim78,Kim86b] appears to be the most general coordination-by-machine approach. The PTC scheme allows the design of error detection and recovery capabilities of a process to be made in a manner independent of recovery structures of other cooperating processes. Moreover, under the PTC scheme, a recoverable region of each process may involve any finite sequence of message exchanges Other coordination-by-machine schemes proposed in the literature, are either incomplete in the sense that they fail to recognize unnecessary RP's or severely restrictive in that they impose severe constraints on message exchanges allowed in a recoverable region.

There are four major elements in the system design

philosophy underlying the PTC scheme.

(a) Independent and structured design of recovery capabilities of processes: The error detection and recovery capabilities of each process are structured with the aid of recovery block and designed in a manner independent of those of other processes.

(b) Transfer of uncommitted messages: A process is allowed to send to other processes information which has not been completely validated. Therefore, the scheme is more flexible than those which require that each message be sent immediately after the test point (TP) of a recovery block execution (RBE) but do not allow message output while a process is in an RBE.

(c) Prohibition of suspicion: Each process is held solely responsible for detecting and correcting errors that it originated. If any process that has failed an acceptance test is allowed to suspect another process of having exported faulty information and demands the exporter to roll back, it is not possible to prevent a domino effect in general [Kim78]. The domino effect in this paper is given a narrow definition of a phenomenon in which a failure of a process at a TP causes the process to make two consecutive rollbacks without performing any useful computation between the two.

(d) Automatic insertion of branch-RP's: Before a process imports uncommitted information, an RP is inserted automatically by the intelligent processor system in order to protect the computational results of the importer process. These RP's, called branch-RP's, are in addition to the RP's established on initiation of RBE's, which are called base-RP's. If the imported information is revoked later, the branch-RP represents the most advanced valid execution point, i.e., the target of the minimum-distance rollback. As shown in [Kim78,Kim86b], the branch-RP's are not necessarily inserted before all import operations involving uncommitted information.

There is a price to be paid for allowing flexible interaction among processes in the PTC/LCN scheme. To be more specific, allowing exchange of incompletely validated information between processes can cause, albeit rare, an undesirable phenomenon called here the exhausted importer (EI) phenomenon. An EI is a process that imported bad information during an RBE and has since exhausted all of its try blocks for no avail in passing the acceptance test.

The most general approach to handling the EI problem which can be cost-effective in most applications seems to be to allow accusations by importer processes in well-defined special situations. The accusations must be allowed in such a manner that the possibility of a domino effect remains non-existent. A reasonable strategy is to allow only the importer processes that have become EIs to accuse the exporters. The purpose here is of course to exploit both the importer's detection capabilities and the exporter's correction capabilities for handling the errors which are propagated from the exporter to the importer and cannot be detected by the exporter.

The PTC scheme is in principle applicable to LAN environments. However, the execution overhead and the size of the worst-case recovery time of the PTC scheme makes it ineffective in applications where application processes are closely coupled and interact frequently. Therefore, the primary application domain of the PTC scheme is in the real-time WAN environments.

In fact, in order to be practical in applications with highly pressing time constraints, the PTC scheme needs to be extended to allow a certain degree of autonomy to each process. To be more specific, when a sender process first sends a message to a receiver process and later revokes it as a part of its rollback, the receiver process may choose one of the following courses of actions under the autonomy-permitting version of the PTC scheme, depending upon the available time and other constraints.

(1) The receiver process ignores the revocation notice as if it has heard nothing.

(2) The receiver process rolls back to an RP established prior to receiving the message that has now been cancelled.

(3) The receiver process takes a compensating action (not an application-independent rollback/undo action) in order to nullify the effect of its own actions based on the message received but later cancelled. This compensating action may take far less time than the rollback-and-retry action does in many situations.

This version of the PTC scheme that allows autonomy to processes is a promising approach to realizing autonomous survival capabilities in real-time WAN environments. As in the case of the conversation scheme, a convincing demonstration of the PTC scheme has yet to take place.

A brief note about the relationship between the DRB scheme discussed in Section 3.7, the conversation scheme discussed in Section 3.9, and the PTC scheme discussed above seems useful here. The DRB scheme mentioned can be used in DCS's to establish the first-level guards for damage confinement and recovery, preventing most of the faults from crossing computing station boundaries. On the other hand, the PTC scheme and conversation scheme can be used to establish the second-level guards, supporting detection and recovery from the faults arising in interaction between computing stations.

4. Major Remaining Issues

This section presents a brief review of some of the issues remaining to be resolved in future research.

(1) Software fault tolerance: design diversity and acceptance test

As pointed out in [Ran75], the redundancy required for making provision for tolerance of software faults is not simple replication of programs but redundancy of design. Although redundant design has been sporadically exploited in the past, it is by and large an unestablished technique. One issue in redundant design is how to maximize the diversity/independence among multiple versions of software, to be more precise, how to minimize the correlated faults present in multiple versions [Avi85,Avi88,Kel88]. Various approaches such as use of different language tools for designing different versions are currently under investigation. Another issue is how to obtain effective acceptance tests capable of determining reasonableness of the computation results at run time without consuming an excessive amount of time. It is desirable to maximize the fault coverage without incurring an unduly complex and time-consuming logic into the acceptance test.

(2) Cooperative recovery from interaction faults

Implicit in the discussions of sections 3.9 and 3.10 is the fact that the degree of coordinating processes during their design for the purpose of effecting coordinated fault detection and recovery actions at run time is a variable parameter. At one extreme, no coordination efforts are made and thus the error detection and backward recovery actions of each process are designed independent of other processes. The PTC scheme is such an approach. However, this approach incurs a high execution (time) cost and excludes the possibility of detecting erroneous behavior of a process by other processes. At the other extreme, processes are tightly coordinated, i.e., cooperative error detection and backward recovery actions of processes are explicitly specified at design time. The conversation scheme supports such an approach. This approach incurs a relatively high design cost. Experiences with both PTC and conversation schemes are lacking. Their efficient implementation requires further studies. In many cases, a middle-road approach positioned between the two extremes may be the most advantageous. Therefore, optimal coordination is another research subject and such research will benefit from, among other things, clear understanding of the powers and costs of the PTC and conversation schemes.

(3) Autonomous and adaptive survival

In many hard-real-time applications, slow recovery is as bad as no recovery. Therefore, it is useful to design processes such that a process can choose to attempt a quick approximate state restoration when there is not enough time to do a complete time-consuming state restoration. The main issue here is how to facilitate systematic design of such processes capable of adaptive recovery. The concept of an autonomy-permitting version of the PTC scheme discussed in Section 3.9 appears to be a good starting point. It seems fair to say that the area of autonomous and adaptive survival is in its infancy.

(4) Validation and evaluation

Some of the schemes discussed in Section 3 have been incorporated into operational systems. However, the data on the improvement of the overall reliability in each system due to the incorporation of the schemes is scarce. This needs to be corrected in order to accelerate the development of the technology for designing fault-tolerant real-time DCS's. After all, the key issue in designing such DCS's is to optimize the performance-reliability tradeoff. In other words, extra computing resources used for fault tolerance could be used for performance enhancement if the system reliability without use of the extra resources is already satisfactory. Without accurate information on the powers and costs of various fault tolerance schemes, the tradeoff cannot be made effectively. On the other hand, some schemes discussed in Section 3 have not even been sufficiently validated, e.g., the conversation scheme, the PTC scheme, etc. Here testbed-based validation is most desirable. The availability of low-cost building-blocks such as microcomputers and interconnection devices, has made the construction of cost-effective DCS testbeds not much more expensive than constructing pure software simulators running on centralized computer systems. Testbeds are capable of representing the operating environment and input scenario more accurately than software simulators. As a result, testbed-based evaluation produces more accurate results than software simulation. Testbed-based evaluation also yields insights into the potential for further optimization of the implementation techniques.

An efficient method for validating the schemes aimed at handling design faults is currently lacking. The main difficulty is in creation of realistic fault conditions.

(5) Integration of fault tolerance schemes

Several fault tolerance schemes discussed in Section 3 complement each other. For example, the complementary relationship among the DRB scheme, the conversation scheme, and the PTC scheme was discussed in Section 3.10. Another interesting example is the relationship between the comparing pair scheme and the DRB scheme. An extension of the DRB scheme in which each of the two nodes (primary and backup) is replaced by a comparing pair for the purpose of enhancing the hardware fault tolerance capability is highly appealing.

Another attractive extension of the DRB scheme (or the conversation scheme or the PTC scheme for that

matter) that should be explored in the future is to provide the capability of repairing a failed node of the DRB station and rejoining the repaired node into the DRB station. The repair here may involve replacing the failed node with a spare node. The rejoin of the repaired node must be accomplished without disrupting the real-time computing service of the DRB station. An extension of the DRB scheme that possesses such non-disruptive rejoin capabilities is called a repairable DRB scheme [Kim88a].

Although the complementary relationship existing among various fault tolerance schemes has been recognized for some time as mentioned above, cost-effective integration of the schemes is largely an unexplored field. Also the scientific foundation for evaluation of the overall cost-effectiveness of a set of fault tolerance schemes is lacking.

5. Summary

In this paper some of the fault tolerance schemes that have been established as promising ones for use in real-time DCS's, have been reviewed. Major issues that remain to be resolved in 1990's have also been discussed. By and large, the design of fault-tolerant real-time DCS's is an immature field. Many of the promising fault tolerance schemes have not been adequately evaluated. It is hoped that much more testbed-based efforts be made in the field of fault-tolerant real-time distributed computing and the issues discussed in Section 4 be resolved in 1990's.

Acknowledgement. This work was supported in part by the National Science Foundation under Grant No. INT-8796259, in part by the Office of Naval Research under Contract No. N00014-87-K-0231, in part by the US Army through NASA JPL, in part by the US Air Force RADC, and in part by a grant from AT&T. The author wishes to acknowledge the help received from S. M. Yang, W. Farrar, and B. J. Min during preparation of this paper.

References

[And75] Anderson, G.A. and Jensen, E.D., "Computer Interconnection Structures: Taxamony, Characteristics, and Examples", ACM Computing Surveys, Dec. 1975, pp.197-213.

[And83] Anderson, T. and Knight, J.C., "A Framework for Software Fault Tolerance in Real-Time System", IEEE TSE, May 1983, pp.355-364.

[Avi85] Avizienis, A., "The N-Version Approach to Fault-Tolerant Software", IEEE Trans. on Software Engineering, Vol. Se-11, No. 12, December 1985, pp.1491-1501.

[Avi88] Avizienis, A., Lyu, M.R., and Schutz, W., "In Search of Effective Diversity. A Six-Language Study of Fault-Tolerant Flight Control Software", Proc. FTCS 18, pp.15-22.

[Bar78] Bartlett, J.F., "A NonStop Operating System", 11th Hawaii Int'l Conf. on System Sciences, Jan. 1978, pp 103-117.

[Bha87] Bhargava, B., editor. "Concurrency and Reliability in Distributed Systems," Van Nostrand and Reinhold, 1987.

[Chu87] Chu, W.W., Kim, K.H., and Mcdonald, W.C., "Testbed-based Evaluation of Design Techniques for Fault-Tolerant Real-Time Distributed Computer Systems", Proceedings of the IEEE, Vol.75, No.5, Special Issue on Distributed Databases, May 1987, pp.649-667.

[Dav80] Davis, C.G. and Couch, R.L., "Ballistic Missile Defense: A Supercomputer Challenge", IEEE Computer, Nov. 1980, pp.37-46.

[For78] Forsdick, H.C., et al., "Operating Systems for Computer Networks", IEEE Computer, Jan. 1978, pp.48-57.

[Fou84] Foudriat, E.C., et al., "An Operating System for future Aerospace Vehicle Computer Systems", NASA Technical Memorandum 85784, April 1984.

[Hec76] Hecht, H., "Fault-Tolerant Software for Real-Time Applications", Computing Surveys Dec. 1976, pp.391-407.

[Hor74] Horning, J.J., Lauer, H.C., Melliar-Smith, P.M., and Randell, B., "A program structure for error detection and recovery", Lecture Notes in Comp. Sci., vol. 16, Springer-Verlag, 1974, pp.171-187.

[Iha84] Ihara, H. and Mori, K., "Autonomous Decentralized Computer Control Systems", Computer, Vol.17, No 8, Aug. 1984, pp.57-66.

[Kat78a] Katsuki, D., et al., "Pluribus - An Operational Fault-Tolerant Microprocessor", Proc. of the IEEE, Oct. 1978, pp.1146-1159.

[Kat78b] Katzman, J.A., "A Fault-Tolerant Computing System", 11th Hawaii Int'l Conf. on System Sciences, Jan. 1978, pp.85-102.

[Kel88] Kelly, J.P.J. et al., "A Large Scale Second Generation Experiment in Multi-Version Software. Description and Early Results", Proc. FTCS-18, pp.9-14.

[Kim78] Kim, K.H., "An Approach to Programmer-Transparent Coordination of Recovering Parallel Processes and Its Efficient Implementation Rules", Proc. 1978 Int'l Conf. on Parallel Processing, August 1978, pp.58-68.

[Kim82] Kim, K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitor", IEEE Trans. on Software Eng., Vol. SE-8, No. 3, May 1982, pp.189-197.

[Kim84] Kim, K.H., "Distributed Execution of Recovery Blocks. an Approach to Uniform Treatment of Hardware and Software Faults", Proc. 4th Int'l Conf. on Distributed Computing System, May 1984, pp.526-532.

[Kim85] Kim, K.H., Yang, S.M., and Kim, M.H., "Implementation of Concurrent Programming Language Facilities Supporting Conversation Structuring", Proc. COMPSAC 85, Oct. 1985, pp.445-453.

[Kim86a] Kim, K.H., Heu, S., and Yang, S.M., "An Analysis of the Execution Overhead Inherent in the Conversation Scheme", Proc. 5th Symp. on Reliability in Distributed Software and Database Systems, Jan. 1986, pp.159-168.

[Kim86b] Kim, K.H., You, J.H., and Abouelnaga, A., "A

Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes", Proc. 16th Int'l Conf. on Fault-Tolerant Computing, July 1986, pp.130-135.
[Kim88a] Kim, K.H. and Yoon, J.C., "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme", Proc. 18th Int'l Symp. on Fault-Tolerant Computing (FTCS-18), pp.50-55.
[Kim88b] Kim, K.H. and Yang, S.M., "An Analysis of the Performance Impacts of Lookahead Execution in the Conversation Scheme", To appear in Proc. 7th Symp. on Reliable Distributed Systems, Columbus, OH, Oct. 1988.
[Kop85] Kopetz, H. and Merker, W., "The Architecture of Mars", Proc. 15th Int'l Symp. on Fault-Tolerant Computing, June 85, pp.274-279.
[Lel81] Le Lann, G., "A Distributed System for Real-Time Transaction Processing", IEEE Computer, Feb. 1981, pp.43-48.
[Lis83] Liskov, Barbara H. and Scheifler, Robert W. "Guardians and Actions: Linguistic Support for Robust Distributed Programs". ACM Transactions on Programming Languages and Systems 5.3 (July 1983), pp.381-404.
[McD82] McDonald, W.C. and Smith, R.W., "A flexible distributed testbed for real time applications", Computer, Vol.15, No.10, Oct. 1982, pp.25-39.
[Ram81] Ramamoorthy, C.V. et al., "Application of a Methodology for the Development and Validation of Reliable Process Control Software", IEEE Trans. on Software Engr., Vol. SE-7, No.6, Nov. 1981, pp.537-555.
[Ran75] Randell, B., "System structure for software fault tolerance", IEEE Trans. on Software Engr., June 1975, pp.220-232.
[Sta84] Stallings, W., "Local Networks", ACM Computing Surveys, March 1984, pp.3-41.
[Str84] 'Stratus Continuous Processing', Stratus Computer, Inc., 1984.
[Toy78] Toy, W.N., "Fault-Tolerant Design of Local ESS Processors", Proceedings of the IEEE, Vol.66, No.10, Oct. 1978, pp.1126-1145.
[Tri83] Tripathi, A.R., and Wang, P.S., "An Object-Oriented Design Model for Reliable Distributed Systems," Proc. Third Symposium on Reliability in Distributed Software and Database Systems, October 1983.
[Wen78] Wensley, J.H., et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", Proc. of the IEEE, Oct. 1978, pp.1240-1255.
[Yau75] Yau, S.S. and Cheung, R.C., "Design of Self-checking Software," Proc. Int'l Conf. on Reliable Software, 1975, pp.450-457.

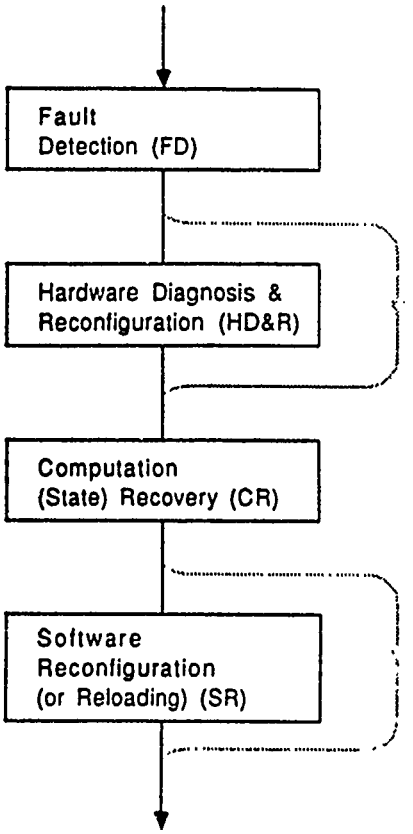


Figure 1. Basic steps in fault tolerance (Adapted from [Kim79])

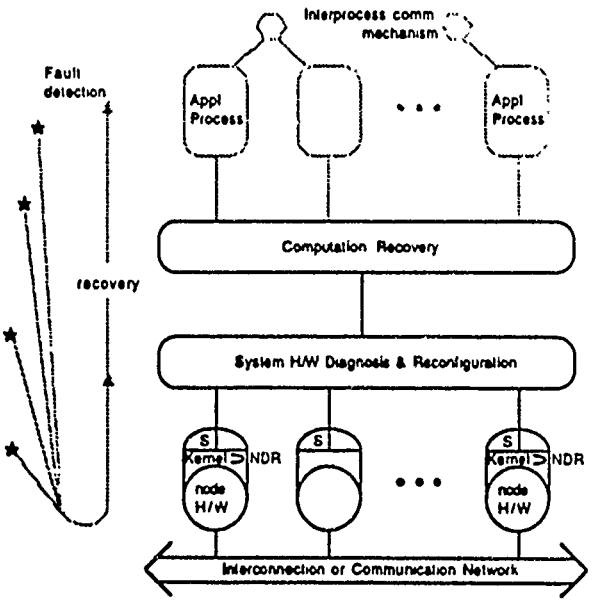


Figure 2. An abstract model of fault-tolerant DCS structure

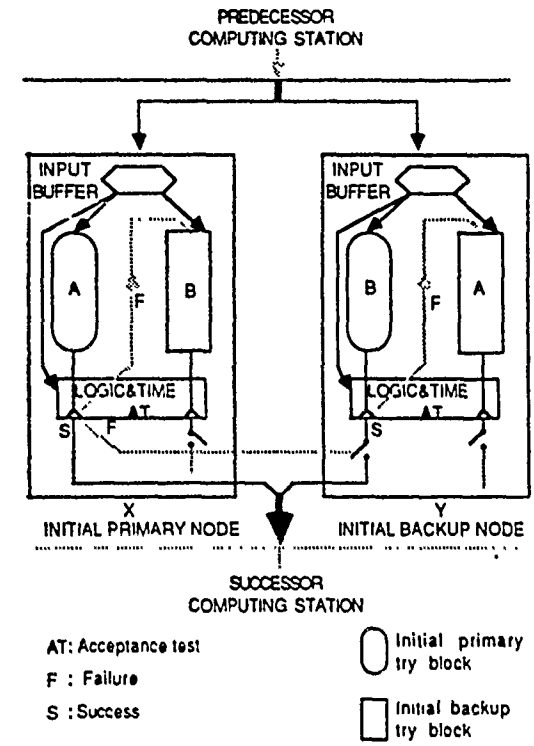


Figure 3. The basic structure of the distributed recovery block (DRB) (Adapted from [Kim88])

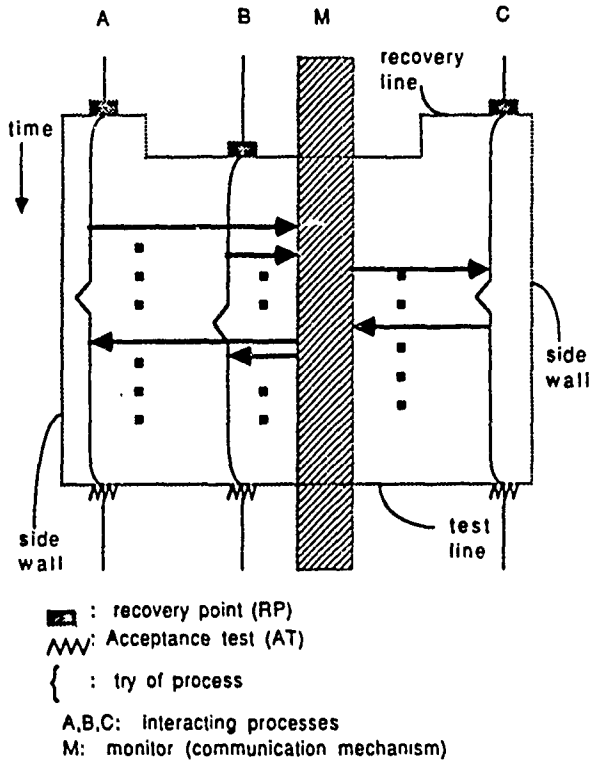


Figure 5. Conversation (Adapted from [Kim82])

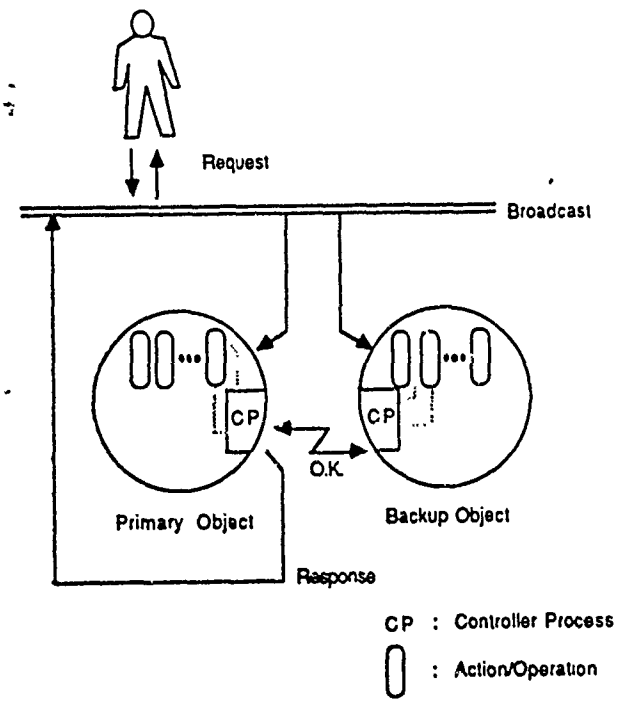


Figure 4. Object replication over LAN based on the DRB principle

Appendix A.II

Approaches for System-Level Fault Tolerance in Distributed Real-Time Computer Systems

Approaches for System-Level Fault Tolerance in Distributed Real-Time Computer Systems

K. H. Kim
Computer Engineering Program, Dept. of Electrical Engineering
University of Calif a, Irvine, CA 92717, USA
(Invited Paper)

ABSTRACT: The purpose of this paper is to summarize major issues in providing the capabilities for tolerance of both hardware faults and software faults in real-time computer systems (DCS's). The paper starts with several guidelines considered to be highly useful in searching for effective system-level fault tolerance schemes. Some promising schemes are then reviewed.

1. Introduction

System-level fault tolerance refers to the continuous operation of a computer system at an acceptable level of performance under occasional presence of faults in its hardware components, operating system, and application software. Historically, hardware faults have received much more serious attention from system designers than software faults have [Avi87]. However, experiences have shown that in many challenging real-time applications, it is infeasible to completely avoid software faults. The types of software faults in real-time computer systems that should be considered include both algorithmic faults and timing faults. In fact, missing deadlines on the part of a real-time computer system is quite often as dangerous as producing incorrect values. Therefore, if incorporation of fault tolerance mechanisms increases the chance of a computer system missing deadlines, it would rather decrease the overall system reliability than improving it.

In designing real-time distributed computer systems (DCS's), system designers must deal with faults in both computing nodes and inter-node connection/communication subsystems. Therefore, the problem of ensuring timely delivery of computation results to the environments has additional dimensions in the case of a DCS, namely, reflection of concurrency and conflicts among distributed nodes as well as inter-node communication delays.

The factors mentioned above make the achievement of system-level fault tolerance in complex real-time DCS's a great challenge to computer system designers and researchers. It seems also easy to forget the fact that fault-tolerant components are desired in building fault-tolerant DCS's but they do not automatically compose the fault-tolerant DCS's. A combination of

in Gorke, W. and Sorensen, H. eds., 'Fault-Tolerant Computing Systems', Informatik-Fachberichte 214, Springer-Verlag, 1989 (Proc. 4th Int'l. Conf. on Fault-Tolerant Computing Systems, Baden-Baden, W. Germany, Sept. 1989) pp. 268-281.

certain components or mechanisms may make the DCS unable to meet deadlines. Needless to say, the composition is also a process prone to logical faults.

The concern with potentially prohibitive costs of various combinations of component-level fault tolerance schemes has led to the search for schemes that can handle in a uniform manner both hardware faults and software faults, the latter including operating system faults and application system faults. This type of schemes are highly desired, considering the potential complexity of large DCS's needed in some real-time applications. Moreover, distinguishing hardware faults from software faults is costly even if it is logically feasible [Kim84]. Also, introducing different treatments for all possible different faults can quickly add up to an intolerable system complexity. In this paper, the discussion will be focused on techniques for unified treatment of both hardware and software faults.

One useful way of classifying application environments of fault-tolerant DCS's is to consider a two-dimensional space in which the "hardness of timing requirements" forms one dimension and the "dirtiness of the environment" forms the other. Based on the hardness of timing requirements, application environments can be roughly classified into real-time applications where deadlines are hard, i.e.,the costs of missing the deadlines are severe, and non-real-time applications. The degree of hardness is in general a continuous variable and thus the boundary between real-time environments and non-real-time environment is somewhat blurred. The dirtiness here refers to the environmental attribute that determines fault rate, including the influence of such factors as electrical noise, physical attacks and vibrations, electromagnetic fields, radiations, dusts, etc.

Typical office environments are examples of clean (low fault rate) environments whereas some space environments and defense environments are the most extreme examples of dirty environments. In this paper, discussions will be mostly related to very dirty environments under very hard timing requirements. System level fault tolerance is much harder to achieve in such environments than in other environments. More importantly, once solutions effective in such dirty hard-real-time environments are obtained, then they can serve as the basis on which suitable solutions for many other real-time environments are derived with relative ease.

In the next section (2), some of the guidelines considered to be highly useful in searching for schemes for system-level fault tolerance are discussed. One of the guidelines discussed is to distinguish between the real-time fault tolerance schemes that can enhance the robustness of a computing station (a processing node executing a single application process) and the supplementary schemes for making a group of cooperating computing stations fault-tolerant Some of the established or promising approaches for "hardening" individual computing stations are discussed in Section 3 whereas some of the promising approaches for hardening computing station groups are discussed in Section 4. The final section (5) summarizes some of the major issues that remain to be resolved.

2. Useful Guidelines in Search for Solutions

Several guidelines considered to be highly useful in searching for effective real-time fault-tolerant distributed computing schemes are presented here.

(1) Search for generic forward recovery schemes

The point here is not so much for favoring forward recovery schemes over backward recovery schemes. It is obvious. In real-time applications, any computation results sent to the environments or the passage of real time can not be recalled. Therefore, there is relatively little room for backward recovery. This is not to say that old state information is not needed in recovery. A design issue here is how and what part of the old state information recorded can be utilized in achieving forward recovery.

A more important keyword here is "generic". For quite some time, many researchers including the author thought that forward recovery, especially recovery from software faults, would have to be "highly" application-dependent. Also, the set of fault conditions to be dealt with would have to be highly limited and a recovery procedure specifically tailored to handling each type of fault condition would have to be provided. Many exception handling approaches that have appeared in literature are examples of highly application-specific forward recovery. Such approaches assume localization of the damage (due to the fault) to a very small area within the system, e.g., one or two program variables. Recently, more generic forward recovery schemes which are capable of dealing with a broader range of fault conditions and more widely spread damages, have emerged, e.g., the distributed recovery block (DRB) scheme [Kim84,Kim88a], the N-version scheme [Avi85], etc. Drastic decrease of the hardware cost occurred in the past decade made users of these schemes largely free of concern in employing twice or three times the amount of hardware used in non-fault-tolerant systems. In the absence of generic forward recovery schemes, the design of fault-tolerant real-time computer systems will remain as a highly artistic error-prone discipline. Some of the promising schemes formulated will be discussed in Section 3. However, much further work is needed in this direction searching for generic forward recovery schemes.

It should be noted, however, that there are certain fault types which dictate recovery procedures more closely tailored to application details than other fault types. The representative example of such a fault is a system-wide disturbance caused by lightning, unreliable power source, radiation, etc., which may be referred to by a generic term temporary blackout [Kim89b]. Efficiency and timing concerns lead to the selection of a procedure for detecting and recovering from temporary blackout that is tailored to the detailed application logic. A typical forward recovery procedure involves first restoring critical state variables with the most recently saved values, next reading the current status of the application environment, and finally establishing the computation to an appropriate state compatible with the current environment [Kim89b].

(2) Search for schemes with bounded recovery time

A relative few fault tolerance schemes that have been proposed for use in DCS's have the desired characteristics of bounded recovery time. Yet this characteristics is so important that schemes without it are not worth c nsidering for use in real-time systems. Of course the upper bound that can be guaranteed must be of reasonable size, e.g., much less than the deadline imposed on the next delivery of a computation result to the environment. If the upper bound on the recovery time is too large, the bounded time characteristics of the recovery procedure is meaningless. Therefore, by checking this single characteristics, most of the seemingly reasonable recovery schemes can be taken out of consideration, thereby allowing us to focus research efforts in narrowly chosen fruitful directions.

(3) Separation of concerns for different network types.

Experiences have shown that attempts to find real-time fault tolerance schemes that can be used commonly in a variety of types of computer networks are mostly unproductive. With respect to achieving system level fault tolerance, there are some fundamental differences in basic characteristics between the real-time local area network (LAN) applications and the real-time wide area network (WAN) applications. For example, inter-node communication costs are very low in the case of real-time LAN's compared to that of real-time WAN's. Similarly, reliability of inter-node communication is much higher or can be made much higher at the same cost in LAN's compared to the case of WAN's. Therefore, location-independent reconfiguration of processes is an affordable and desirable capability in real-time LAN's whereas it is not a sensible approach to pursue in real-time WAN's such as WAN's embedded in multiple cooperating satellites. Closely coordinated and highly interactive fault detection and recovery actions can also be designed into real-time LAN's at reasonable costs whereas such attempts will produce negative results in real-time WAN's. A certain degree of node autonomy in both application processing and fault handling is a desirable characteristics in real-time WAN's but the node autonomy is much less important in real-time LAN's.

Fundamental differences also exist between real-time LAN's and real-time tightly coupled networks (TCN's) in which multiple processing nodes communicate via shared memory or high-speed parallel bus. TCN's are used primarily for reasons of high throughput and fast turnaround whereas LAN's are used primarily for reasons of modular expansion, resistance to location-dependent faults, and employment of multiple distributed sensors. Therefore, attempts to develop schemes for system level fault tolerance that can be commonly effective in both real-time LAN's and real-time TCN's are not likely to be very productive, given the current limited understanding of techniques suitable for each network type.

(4) Treatment of faulty process interaction as a second-order problem

Detection of and recovery from faults propagated between cooperating processes is in general a problem that has been attacked for about 15 years but still remains largely unresolved [Ran75,Kim78]. In the context of an abstract DCS model consisting of a network of computing stations, each of which is a subsystem of the DCS and executes a single application process,

this problem is equivalent to how to make a group of cooperating computing stations fault tolerant. In real-time applications, it is hopeless to handle this faulty interaction problem unless the frequency of their occurrence is below a certain threshold. In fact, most of the complex approaches that have been proposed are unsuitable because of their poor bounded recovery time characteristics. The most obvious yet important way to minimize the occurrence of faulty interaction is to make each computing station highly unlikely to leak faulty information to others. It seems sensible to deal with the faulty interaction problem only after incorporating effective schemes for making individual computing stations fault-tolerant, especially in real-time applications. In other words, the faulty process interaction problem should be viewed as a second-order problem and the schemes mobilized for handling it viewed as supplements to those for "hardening" individual computing stations. For example, if we can assume that the primary schemes handle the transient faults of computing station hardware and ensure timely, orderly, and reliable transmission of messages between computing stations, then the problem of finding cost-effective schemes for handling faulty process interaction becomes much more feasible. Furthermore, if it is feasible to design each computing station to perform thorough validation of each message to be sent to other computing stations, the faulty interaction problem becomes substantially simplified.

3. System-level Fault Tolerance Schemes for Hardening Individual Computing Stations

Schemes for handling hardware faults only have been extensively developed [Car85,Toy87]. In this section, the discussion will be focused on the schemes that can handle both hardware faults and software faults. All the schemes to be discussed in this section are based on the following concepts.

(1) Multiple versions of computing components

The approach of using two or more copies of a hardware component is a well-established technique for hardware fault tolerance [Hop78,Toy78]. In early 70's, the underlying concept of replicative redundancy was extended into the software arena, resulting in the approach of using multiple functionally equivalent or similar versions of a software component [Hor74]. Therefore, in this approach multiple design efforts are made with the expectation that the same type of design fault will not be introduced into all the versions. To achieve this goal of avoiding common faults, techniques for maximizing the diversity in the algorithms, the design methods, and the design tools used for developing multiple versions, have been explored [And81,Avi88].

(2) Acceptance test vs. voting

There are two fundamentally different approaches to determining the quality of the execution results of multiple versions. One is to design another software component called the acceptance test which is an expression of the acceptability criterion that every version is

required to meet [Hor74]. This acceptance test is executed to determine the acceptability of each version. The other approach is to compare the execution results of multiple versions [Avi85]. If the number of versions is three or more, then majority-voting can be used to recognize the result to be adopted, thereby facilitating recovery.

While the acceptance test approach requires additional design effort, it facilitates recovery with only two versions; the first result passing the acceptance test will be adopted. It also allows use of two versions which are similar but not functionally equivalent. For example, one version may be designed to produce the minimum acceptable results while the other may be designed for more desirable results (e.g., more accurate results, better optimized results, etc.). Moreover, the voting approach requires design of multiple versions expected to generate truly identical computation results. This could be a severe restriction in cases where complexity of a program component is high. The voting approach also requires close synchronization of the nodes executing multiple versions. On the other hand, the voting process itself does not require any help from the application developer while it requires at least three versions of the application software component.

When the "fail-stop" is one of the intended final states of a software component, two versions supported by a comparison mechanism would be sufficient. However, the recovery responsibility then rests outside that software component.

In principle, various combinations of the acceptance test approach and the voting approach are conceivable. Effectiveness of such approaches is largely unknown at present.

3.1 The distributed recovery block (DRB) scheme

The DRB scheme depicted in Figure 1 [Kim84,Kim88a,Kim89a] is essentially an active redundancy scheme where multiple processors concurrently execute multiple versions of a software component and then the same acceptance test. Each processor uses a time-out mechanism, as well. An important advantage of this scheme is that fast forward recovery can be achieved with the software redundancy produced through the aid of a convenient design tool, i.e., the recovery block [Hor74,Ran75].

Recovery block consists of one or more routines, called try blocks here, designed to compute the same or similar results, and an acceptance test. For the sake of simplicity in description, a recovery block is assumed to contain only two try blocks, i.e., the primary and the alternate. While the original recovery block scheme incorporated the backward recovery approach, the DRB scheme incorporated a new forward recovery capability.

The basic idea of the DRB scheme is to use two processor nodes to execute two different try blocks and then the same acceptance test concurrently. Both nodes use watchdog timers. After receiving the common input data from the predecessor computing station, each node executes a try block and then the acceptance test. If the node running the primary try block

passes the test in time, it sends a signal "I'm OK" to the backup node (running the alternate try block). It then outputs the results to the successor computing station. As long as the primary node sends the signal to the backup node in time, the latter suppresses its own results. Therefore, only if the primary node dies or fails to pass the acceptance test, the backup node will start outputting its results.

This approach has two useful characteristics:
(a) Recovery can be accomplished in the same manner regardless of whether a node fails due to a hardware fault or a software fault, i.e., it is unnecessary to distinguish between hardware faults and software faults;
(b) The recovery time is minimal since maximum concurrency is exploited between the primary and the backup nodes.

The experimental study reported in [Kim89a] has clearly demonstrated high-speed recovery capability of this scheme. More extensive experiments are under way in multiple locations [Hec87].

The scheme can thus be used to obtain highly reliable computing stations each dedicated to execution of a specific real-time (atomic) task.

3.2 The DRB scheme combined with the comparing pair scheme

The comparing pair scheme under which hardware components such as processors and memory modules are duplicated and their outputs are compared is an effective fault detection scheme widely used [Toy78, Str84]. An extension of the DRB scheme in which each of the two nodes (primary and backup) is replaced by a comparing pair for the purpose of enhancing the hardware fault tolerance capability is highly appealing. In such an extended scheme, most of the hardware faults will be detected by the acceptance test and/or other mechanisms built into the node pair.

3.3 The voting N-version scheme

This scheme uses multiple versions with the voting approach for fault detection and result selection. As the use of two versions in the DRB scheme is the standard practice due to the high cost of designing and maintaining multiple versions, the use of three versions is the standard practice in the case of the voting N-version scheme. Some experimental studies have shown the plausibility of achieving software fault tolerance by use of the scheme [Avi88,Kel88] Due to the restrictive nature of the voting approach discussed earlier, the application domain of the voting N-version scheme will be somewhat narrower than that of the DRB scheme. There have been proposals for removing this restriction by combining the voting approach with the common acceptance test and other evaluation approaches. Such extensions are currently at the concept development stage.

passes the test in time, it sends a signal "I'm OK" to the backup node (running the alternate try block). It then outputs the results to the successor computing station. As long as the primary node sends the signal to the backup node in time, the latter suppresses its own results. Therefore, only if the primary node dies or fails to pass the acceptance test, the backup node will start outputting its results.

This approach has two useful characteristics:
(a) Recovery can be accomplished in the same manner regardless of whether a node fails due to a hardware fault or a software fault, i.e., it is unnecessary to distinguish between hardware faults and software faults;
(b) The recovery time is minimal since maximum concurrency is exploited between the primary and the backup nodes.

The experimental study reported in [Kim89a] has clearly demonstrated high-speed recovery capability of this scheme. More extensive experiments are under way in multiple locations [Hec87].

The scheme can thus be used to obtain highly reliable computing stations each dedicated to execution of a specific real-time (atomic) task.

3.2 The DRB scheme combined with the comparing pair scheme

The comparing pair scheme under which hardware components such as processors and memory modules are duplicated and their outputs are compared is an effective fault detection scheme widely used [Toy78, Str84]. An extension of the DRB scheme in which each of the two nodes (primary and backup) is replaced by a comparing pair for the purpose of enhancing the hardware fault tolerance capability is highly appealing. In such an extended scheme, most of the hardware faults will be detected by the acceptance test and/or other mechanisms built into the node pair.

by the comparing pair mechanism and some of the remaining hardware and software faults will be detected

3.3 The voting N-version scheme

This scheme uses multiple versions with the voting approach for fault detection and result selection. As the use of two versions in the DRB scheme is the standard practice due to the high cost of designing and maintaining multiple versions, the use of three versions is the standard practice in the case of the voting N-version scheme. Some experimental studies have shown the plausibility of achieving software fault tolerance by use of the scheme [Avi88,Kel88]. Due to the restrictive nature of the voting approach discussed earlier, the application domain of the voting N-version scheme will be somewhat narrower than that of the DRB scheme. There have been proposals for removing this restriction by combining the voting approach with the common acceptance test and other evaluation approaches. Such extensions are currently at the concept development stage.

3.4 Major remaining issues

Of many issues remaining unresolved in this area, two most important ones are as follows.

(1) Validation of the software fault tolerance capability of the multiple version approach.

Although redundant design has been sporadically exploited in the past and research efforts in this area have been stepped up in recent years, it is by and large an unestablished technique. The critical question here is : can this approach truly handle the faults that were not foreseen at the design and testing time? As far as run-time detection of such faults is concerned, there have been a small number of success reports [Hag87]. However, convincing demonstrations of real-time recovery from such faults by use of a multiple version approach have yet to be seen.

(2) Non-disruptive rejoin of repaired nodes.

An attractive extension of the DRB scheme or the N-version scheme that should be explored in the future is to provide the capability of repairing a failed node of the fault-tolerant computing station (running under the DRB scheme or the N-version scheme) and rejoining the repaired node into the computing station. The repair here may involve replacing the failed node with a spare node. The rejoin of the repaired node must be accomplished without disrupting the real-time computing service of the computing station [Kim88a].

4. Some Promising Approaches for Hardening Computing Station Groups

For handling hardware faults only the problem is considerably simpler. Techniques such as the comparing pair scheme can be used to reduce the probability of faulty information crossing the computing station boundary to a negligible extent. Efficient techniques have also been developed for establishment of recovery lines, i.e., coordinated sets of recovery points of computing stations that define consistent system states [Ran75,Kim82], which are needed to deal with temporary blackout events [Ton89].

Real challenges are again in achieving real-time forward recovery from both hardware faults and software faults. With software faults present the probability of faulty information crossing the computing station boundary can no longer be ignored. First of all, even if a computing station is designed to perform a priori validation of each message sent to other stations, such validation can provide only limited coverage in detection of erroneous message contents. Secondly, execution time costs and design complexity considerations often lead to design of capabilities for validation of a computation segment that produces multiple messages (at variable intervals) for other computing stations rather than for a computation segment that produces only one message at the end. A computation segment combined with a validation and

recovery capability in a computing station is called here a recoverable region. Therefore, a recoverable region often sends messages to other computing stations and must revoke the messages if the validation performed at its exit point results in a failure.

In the presence of the non-negligible probability of faulty information being propagated between computing stations, the stations must be designed to cooperate in recovery. If the recovery-related actions of computing stations are not well coordinated, then situations where recovery costs become prohibitive can develop. A well-known example of such a station is where a domino effect, which is a cyclic chain of rollback propagations among computing stations, occurs due to the uncoordinated nature of the recovery points established by the stations [Ran75].

Another aspect worth noting here is that there are fundamental differences between the recovery lines needed in dealing with hardware faults and those for software faults. Recovery lines for hardware faults may be established in a manner independent of the application software structure. For example, clock-driven establishment is feasible as long as fixed recovery procedures applicable regardless of their invocation time are used. On the other hand, in the case of software faults, no meaningful recovery procedures which can be invoked at any time are conceivable. Only the recovery procedures which are designed to retry the computation segments under suspicion from specific execution points in the computation structure with different versions have reasonable chances of producing good results.

Fault tolerance schemes for hardening groups of computing stations can be classified on the basis of the degree of coordinating processes (to run on different stations) during their design for the purpose of effecting coordinated fault detection and recovery actions at run time. At one extreme, no coordination efforts are made and thus the error detection and recovery actions of each process are designed independent of other processes. The programmer-transparent coordination (PTC) scheme to be discussed later in this section is such an approach. It requires an "intelligent" system kernel (or a virtual machine) which automatically establishes appropriate recovery points of interacting processes. However, this pure "run-time coordination" approach incurs a high execution (time) cost and excludes the possibility of detecting erroneous behavior of a process by other processes. Also, the option of trying alternate interaction scenarios following a fault detection is not available to cooperating processes since processes are not produced via such coordinated design efforts. At the other extreme, processes are tightly coordinated, i.e., cooperative error detection and recovery actions of processes are explicitly specified at design time. The conversation scheme to be discussed in this section supports such a "design-time coordination" approach. This approach incurs a relatively high design cost. Many middle-road approaches positioned between the two extremes are conceivable and may be highly advantageous in some application environments.

4.1 The distributed conversation (DCONV) scheme

The distributed conversation (DCONV) scheme is intended for use in real-time tightly coupled network (TCN) or local area network (LAN) applications where it is natural to design a group of computing stations together in a closely coordinated manner. The scheme is meant to be a practical version based on the abstract structure proposed by Randell [Ran75].

The abstract conversation structure proposed by Randell is a conceptual extension of the recovery block which is a concrete language construct devised to support structuring the error detection and recovery actions of a single process. It is a two-dimensional enclosure of recoverable activities of multiple interacting processes, i.e., recoverable interacting session [Kim82,Ran75]. The enclosure consists of a recovery line, a test line, and two side walls defining exclusive membership as shown in Figure 2. A recovery line is a coordinated set of the recovery points of interacting processes that are established (possibly at different times) before interaction begins. When all the processes roll back to the recovery line, there cannot be any further propagation of rollbacks among processes. A test line is a correlated set of the acceptance tests of the interacting processes. A conversation is successful only if all the interacting processes pass their acceptance tests forming the test line. If any of the acceptance tests fails, all the processes roll back to the recovery line and retry with their alternate try blocks. These alternate try blocks collectively define an alternate interacting session, whereas the set of primary try blocks executed first after the processes enter the conversation define the primary interacting session. Therefore, processes cooperate in error detection, regardless of the source of the error. Two sidewalls defined by a conversation imply that the processes participating in the conversation must neither obtain information from nor leak information to a process not participating in the conversation. That is, no "information smuggling" by processes in a conversation is permitted.

Various research groups have invested their efforts toward the goal of converting the abstract concept described above into a technology usable by practitioners. Some of the major results obtained include practical language tools for conversation design [Gre85,Kim85], understanding of the execution overhead inherent in the conversation scheme [Kim86a], and approaches to implementation of the conversation scheme in LAN's [Yan89].

The conversation scheme as described above incorporates the backward recovery approach and thus is not suitable for many real-time applications. However, a conceptually clear approach that solves this problem can be borrowed from the DRB scheme [Kim84,Kim89a], namely, parallel execution of primary and alternate interacting sessions. This means that the primary interacting session runs on one group of interconnected computing nodes and the backup interacting session runs on another group of computing nodes. Timeout mechanisms are used to ensure timely execution of interacting sessions. The resulting scheme was termed the distributed conversation (DCONV) scheme. Implementation techniques for the DCONV scheme have not yet been studied to depth.

The DCONV scheme goes beyond the DRB scheme in that the former can handle interaction faults. Both schemes are aimed at tight error containment by defining rigid

enclosures for possibly faulty activities. In spite of the recognized potential of the DCONV scheme, a convincing demonstration of the utility of the scheme in real-time applications has yet to take place.

4.2 The programmer-transparent coordination with obedient receivers (PTC/OR) scheme

Unlike the DCONV scheme, the PTC scheme is intended for use in real-time WAN (or some LAN) applications where it is desirable to build some autonomy into each of the distributed nodes due to relatively expensive and unreliable inter-node communication, security concerns, etc. [Kim86b]. The PTC scheme imposes no requirements on coordinated design of fault detection and recovery actions of interacting real-time distributed processes. The PTC scheme allows the design of fault detection and recovery capabilities of a process to be made in a manner independent of recovery structures of other cooperating processes. Also, a recoverable region of each process may involve any finite sequence of message exchanges.

The essence of the PTC scheme is to facilitate run-time coordination of independently designed processes for their cooperative error detection and recovery with the aid of an intelligent virtual machine. The intelligent virtual machine must be capable of supporting the processes in such a way that the domino effect does not occur. There are four major elements in the system design philosophy underlying the PTC scheme.

(1) Independent and structured design of recovery capabilities of processes.
(2) Transfer of uncommitted messages: A process is allowed to send to other processes information which has not been completely validated.
(3) Prohibition of suspicion: Each process is held solely responsible for detecting and correcting errors that it originated.
(4) Automatic insertion of branch recovery points: Before a process imports uncommitted information, a recovery point called a branch recovery point is inserted automatically by the intelligent virtual machine in order to protect the computational results of the importer process

As shown in [Kim78,Kim86b], the branch recovery points are not necessarily inserted before all import operations involving uncommitted information. The protocols for cooperation among distributed intelligent kernels that are optimal in the sense that they incur minimal overhead in both state saving and recovery, have been developed [Kim86b,Kim88b].

Once a process fails in a self-validation at the end of a recoverable region, it must notify all the processes that have received messages that it sent out from within the recoverable region. In effect the messages are revoked. Each notified process must "cleanse" itself of the effects of the messages revoked. A conceptually straightforward way is to roll back to the most recent recovery point established before receiving the messages and then restart. A version of the PTC scheme that uses this backward recovery action is called the PTC with obedient receivers (PTC/OR) scheme.

When the probability of an erroneous message being sent out from within a recoverable region is below a certain threshold, this PTC/OR scheme may exhibit acceptable performance in many real-time WAN environments. However, the worst-case recovery time characteristics of the PTC/OR scheme is very poor. In fact, permission of exchange of incompletely validated information between processes coupled with prohibition of receiver processes from suspecting sender processes can cause, albeit rare, an undesirable phenomenon called here the exhausted importer (EI) phenomenon [Kim86b]. An EI is a process that imported bad information while inside a recoverable region and has since exhausted all of its alternate versions for no avail in passing the acceptance test. The EI phenomenon can make the recovery time exceed the tolerable limit in many real-time WAN environments. Therefore, the PTC/OR scheme may be attractive in real-time WAN applications where the timing requirements are not very hard but it is not suitable for use in hard-real-time WAN applications. In spite of its potential, a demonstration of the PTC/OR scheme in realistic application contexts has yet to take place.

4.3 The programmer-transparent coordination with adaptive receivers (PTC/AR) scheme

The PTC with adaptive receivers (PTC/AR) scheme is a version of the PTC scheme devised for use in applications with hard timing requirements [Kim88c]. The PTC/AR scheme allows a certain degree of autonomy/adaptiveness to each process in cleansing itself of the effects of the messages revoked. To be more specific, when a sender process first sends a message to a receiver process and later revokes, the receiver process may choose one of the following courses of actions under the PTC/AR scheme, depending upon the available time and other constraints.

(1) The receiver process ignores the revocation notice as if it has heard nothing.
(2) The receiver process rolls back to a recovery point established prior to receiving the message that has now been cancelled.
(3) The receiver process takes a compensating action (not an application-independent rollback/undo action) in order to nullify the effect of its own actions based on the message received but later cancelled. This compensating action may take far less time than the rollback-and-retry action does in many situations.

Action (2) is the only option under the PTC/OR scheme. Even in the case of taking action (2), subsequent actions under the PTC/AR scheme need not be retries of the same actions taken before the rollback, thus differing from the actions that may be taken under the PTC/OR scheme.

The PTC/AR scheme provides a framework in which both backward and forward recovery capabilities can be adaptively employed. In the case where processes take backward recovery actions, the consistency among their actions is ensured by the operating rules and built-in mechanisms of the PTC scheme. Therefore, system designers can focus on verification of forward recovery actions of processes. However, provision of all adaptive recovery actions

could present considerable challenge to system designers. Development of system aids that are helpful in such design efforts is an important topic for future research.

In spite of the sound logical basis and the promising nature of the PTC/AR scheme for use in real-time WAN's, the scheme is an unproven technology at an early stage of development.

4.4 Major remaining issues

Due to the relative little understanding obtained regarding the problems encountered in hardening computing station groups in real-time DCS's, the issues that remain unresolved are numerous. Of these, the most pressing one seems to be to determine how to allocate design efforts and run-time resources between hardening individual computing stations and hardening computing station groups. Such decision making requires good understanding of the costs and benefits of various system-level fault tolerance schemes. The knowledge available at present represents a small fraction of what is needed.

5. Summary

Achieving system-level fault tolerance in challenging real-time DCS's is still a distant goal Handling of software faults is largely a technology in its infancy. Many of the promising fault tolerance schemes, in particular, those aimed at hardening computing station groups have not been adequately evaluated. In order to achieve significant advances in this field, much more testbed-based experimental efforts (e.g., [Chu87]) must be made.

In many process control applications, one does not have to make pessimistic assumptions about the possible eruption of software faults. Therefore, most of the approaches discussed in this paper would be too expensive for use in such applications. However, it is believed that simplified versions of the approaches which will be cost-effective in such applications can be derived without much difficulty.

Acknowledgements: This work was supported in part by the Office of Naval Research under Contract N00014-87-K-0231, in part by the University of California MICRO Program and AT&T under Grant 88-123 and in part by NASA JPL and the U.S. Army SDC under Contract NAS7-913-RE-182/443.

References

[And81] Anderson, T. and Lee, P.A., 'Fault Tolerance: Principles and Practice', Prentice-Hall Int'l, Inc., London, 1981.

[Avi85] Avizienis, A., "The N-Version Approach to Fault-Tolerant Software", IEEE Trans on Software Engineering, Vol. Se-11, No. 12, December 1985, pp.1491-1501.

[Avi87] Avizienis, A., Kopetz, H., and Laprie, J.C. eds., 'The Evolution of Fault-Tolerant Computing', Springer-Verlag, New York, 1987.
[Avi88] Avizienis, A., Lyu, M.R., and Schutz, W., "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software", Proc. FTCS-18, pp.15-22.
[Car85] Carter, W.C., "Hardware Fault Tolerance", Chapter 2 in Anderson, T., ed., 'Resilient Computing Systems', Vol. 1, Wiley-Interscience, 1985, pp.11-63.
[Chu87] Chu, W.W., Kim, K.H., and Mcdonald, W.C., "Testbed-based Evaluation of Design Techniques for Fault-Tolerant Real-Time Distributed Computer Systems", Proceedings of the IEEE, Vol.75, No.5, Special Issue on Distributed Databases, May 1987, pp.649-667.
[Gre85] Gregory, S.T. and Knight, J.C., "A new Linguistic Approach to Backward Error Recovery", Proc. FTCS-15, 1985, pp.404-409.
[Hag87] Hagelin, G., "ERICSSON Safety System for Railway Control", in U. Voges ed., 'Software Diversity in Computerized Control Systems', Springer Verlag, Vienna, 1987, pp.11-21.
[Hec87] Hecht, M., Hochhauser, So, and Hecht, H., "Extended Distributed Recovery Blocks for Nuclear Reactor Control and Safety Functions," Final Report, Contract DE-AC03-87-ER80532, Dec. 87.
[Hop78] Hopkins, A.L., et al., "FTMP--A highly Reliable Fault-Tolerant Multiprocessor for Aircraft", Proc. IEEE, Vol. 66, No. 10, Oct. 1978, pp.1221-1239.
[Hor74] Horning, J.J., Lauer, H.C., Melliar-Smith, P.M., and Randell, B., "A program structure for error detection and recovery", Lecture Notes in Comp. Sci., vol. 16, Springer-Verlag, 1974, pp.171-187.
[Kel88] Kelly, J.P.J. et al., "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", Proc. FTCS-18, pp.9-14.
[Kim78] Kim, K.H., "An Approach to Programmer-Transparent Coordination of Recovering Parallel Processes and Its Efficient Implementation Rules", Proc. 1978 Int'l Conf. on Parallel Processing, August 1978, pp.58-68.
[Kim82] Kim, K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitor", IEEE Trans. on Software Eng., Vol. SE-8, No. 3, May 1982, pp.189-197.
[Kim84] Kim, K.H., "Distributed Execution of Recovery Blocks: an Approach to Uniform Treatment of Hardware and Software Faults", Proc. 4th Int'l Conf. on Distributed Computing System, May 1984, pp.526-532.
[Kim85] Kim, K.H., Yang, S.M., and Kim, M.H., "Implementation of Concurrent Programming Language Facilities Supporting Conversation Structuring", Proc. COMPSAC 85, Oct. 1985, pp.445-453.
[Kim86a] Kim, K.H., Heu, S., and Yang, S.M., "An Analysis of the Execution Overhead Inherent in the Conversation Scheme", Proc. 5th Symp. on Reliability in Distributed Software and Database Systems, Jan. 1986, pp.159-168.
[Kim86b] Kim, K.H., You, J.H., and Abouelnaga, A., "A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes", Proc. 16th Int'l Conf. on Fault-Tolerant Computing, July 1986, pp.130-135.
[Kim88a] Kim, K.H. and Yoon, J.C., "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme", Proc. 18th Int'l Symp. on Fault-Tolerant Computing (FTCS-18), pp.50-55.
[Kim88b] Kim, K.H., "Programmer-Transparent Coordination of Recovering Concurrent Processes: Philosophy and Rules for Efficient Implementation", IEEE Trans. on Software Engr., Vol.14, No.6, June 1988, pp.810-821.
[Kim88c] Kim, K.H., "Designing Fault Tolerance Capabilities into Real-Time Distributed Computer Systems", Proc. IEEE Computer Society's Workshop on Future Trends of Distributed Computing Systems in the 1990s, Sept. 1988, Hong Kong, pp.318-328.
[Kim89a] Kim,K.H. and Welch, H.O., "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications", IEEE Trans. on Computers, Vol.38, No.5, May 1989, pp.626-636.
[Kim89b] Kim, K.H., "An Approach to Experimental Evaluation of Real-Time Fault-Tolerant Distributed Computing Schemes", IEEE Trans. on Software Engineering, Vol. 15, No. 6, June 1989, pp.715-725.
[Ran75] Randell, B., "System structure for software fault tolerance", IEEE Trans. on Software Engr., June 1975, pp.220-232.
[Str84] 'Stratus Continuous Processing', Stratus Computer, Inc., 1984.
[Ton89] Tong, Z., Kain, R.Y., and Tsai, W.T., "A Loosely Synchronized Checkpointing Scheme for Rollback Recovery in Distributed Systems", Tech. Report, TC-DS-13, Dept. of Electrical Engineering, Univ. of Minnesota, Minneapolis, MN 55455.
[Toy78] Toy, W.N., "Fault-Tolerant Design of Local ESS Processors", Proceedings of the IEEE,

Appendix A.III

Approaches to Implementation of a Repairable Distributed Recovery Block Scheme

Approaches to Implementation of a Repairable Distributed Recovery Block Scheme

K.H. Kim and J.C. Yoon

Computer Engineering Program, Dept. of Electrical Engineering
University of California Irvine, Calif. 92717

Abstract

The basic concept of the distributed recovery block (DRB) scheme was proposed earlier as an approach to uniform treatment of hardware and software faults in real-time applications. Design issues that arise in implementing the DRB scheme are discussed in this paper together with some promising approaches. Issues in extending the DRB scheme with the capability of reincorporating a repaired node without disrupting the real-time computing service are also discussed. An experimental implementation of the repairable DRB scheme into a real-time distributed computer system (DCS) testbed and subsequent measurement of the system performance demonstrated the fast forward recovery capability and the logical soundness of the scheme.

Index Words: distributed recovery block, forward recovery, real-time recovery, non-disruptive rejoin, acceptance test, timeout

1. Introduction

Many challenging real-time applications require computer systems capable of delivering outputs within the tightly specified deadlines in spite of occasional occurrences of their component malfunctions [Hec76,McD82]. As the service demands imposed on such real-time computer systems continue to increase, the structure of the systems is becoming increasingly more complex, often involving hundreds or even higher numbers of microcomputers. The complexity of software is also considerable in such systems. Therefore, it has become infeasible to completely avoid software faults in such environments. Moreover, distinguishing hardware faults from software faults in such environments is difficult and costly even if it is logically feasible. Often the symptoms of transient hardware faults and those of software faults are similar. Therefore, a technique that can treat both hardware and software faults in a uniform manner has become a highly desired subject.

The basic concept of the distributed recovery block (DRB) scheme was proposed in [Kim84] as an approach to uniform treatment of hardware and software faults in real-time applications. An attractive characteristics of this scheme is the fast forward recovery effect that it produces while supporting systematic incorporation of software redundancy. Although the concept initially appeared to be simple, it has been learned through subsequent experimental efforts that there are many design parameters that have to be chosen carefully in implementing the DRB scheme in each given environment in order to realize the full potential of the scheme.

The purpose of this paper is as follows.
(1) Design issues that arise in implementing the DRB scheme are discussed together with some approaches formulated since the initial conception of the scheme.
(2) An attempt has also been made to extend the original DRB scheme with the capability of reincorporating a repaired node into a computing station without disrupting the real-time computing service of the station. Such an extended DRB scheme is called a repairable DRB scheme in this paper. The issues in facilitating such non-disruptive rejoin of repaired nodes are discussed together with some approaches studied.
(3) In order to validate a repairable DRB scheme, an experimental implementation of the scheme has been carried out by using the real-time distributed computer system (DCS) testbed facilities established in the authors' institution. The implementation structure and the performance data measured are presented.

Section 2 provides an overview of the basic principles of the DRB scheme. Implementation issues and approaches are discussed in section 3 while section 4 deals with the issues in design of non-disruptive rejoin capabilities. An experimental validation of the scheme that has been conducted is sketched in section 5. The final section provides a brief summary.

2. Basic Principles of the Distributed Recovery Block (DRB) Scheme

The DRB (distributed recovery block) scheme is based on a combination of both the distributed processing and the recovery block structuring concepts. Recovery block [Hor74,Ran75] is a language construct supporting the incorporation of program redundancy into a fault-tolerant program in a concise and easily readable form. The syntax of recovery block is as follows: **ensure** T **by** B1 **else by** B2 ... **else by** Bn **else error.** Here, T denotes the acceptance test (AT), B1 the primary try block, and Bk, $2 \leq k \leq n$, the alternate try blocks. All the try blocks are designed to produce the same or similar computational results. The acceptance test is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A try (i.e., execution of a try block) is thus always followed by an acceptance test. In a sense, recovery block is an enclosure of some recoverable activities of a process.

The real-time DCSs considered here are assumed to have the following characteristics:
1) A DCS consists of multiple computing stations, each executing one and only one recovery block.
2) The result produced from a computing station may become an input to another computing station or to the application environment.

3) A computing station may consist of one or more computing nodes. Multiple computing nodes within a computing station can be used either in a load-sharing or in a redundant processing mode.

The DRB scheme exploits concurrent execution of try blocks to facilitate fast forward recovery. For simplicity only two try blocks in a recovery block, the primary and the backup, are used to illustrate the DRB scheme. The specification of the maximum execution time allowed for each try block is an integral part of the DRB scheme [Hec76,Kop85]. A try not completed within the time due to hardware faults or excessive looping is treated as a failure. Therefore, the acceptance test can be viewed as a combination of both logic and time acceptance tests.

The DRB scheme realized with two nodes is depicted in Figure 1. Both primary and backup nodes contain the same acceptance test, consisting of logic and time tests, and the same set of try blocks, A and B. However, the roles of the two try blocks are assigned differently in the two nodes. Primary node X uses try block A as the primary try block initially, whereas backup node Y uses try block B as the initial primary. Therefore, until a fault is detected, both nodes receive the same input data, process the data by use of two different try blocks (i.e., block A on node X and block B on node Y), and check the results by use of the acceptance test. Both nodes perform all these tasks concurrently. The time acceptance test is used to ensure timely behavior of both nodes.

In a fault-free situation, both nodes will pass the acceptance test with the results computed with their primary try blocks. In such a case, the primary node notifies the backup of its success in the acceptance test. Thereafter, only the primary node sends its output to the successor computing station. However, if the primary node fails and the backup node passes its test, the backup node assumes the role of the primary, i.e., the nodes exchange their roles. To be more specific, the primary node attempts to inform the backup node upon its failure in passing the acceptance test. The backup node will take over the role of the primary as soon as it receives notice. If the primary node is completely lost, the backup node will recognize the failure of the primary upon expiration of the preset time limit. It will then become the new primary. Since these interactions between two nodes are done asynchronously, the scheme does not suffer from synchronization overhead. On the other hand, if the backup node fails first, the primary node need not be disturbed. In both cases, the failed node attempts to serve as a backup node; it attempts to roll back and retry with its alternate try block to bring its application computation state or local database up-to-date. This attempt does not disturb the primary node.

Under the DRB scheme, the recovery time is minimal because maximum concurrency is exploited in the redundant try block execution. Fast forward recovery is achieved regardless of whether faults occur in the hardware or software components.

3 Approaches to Implementation of the DRB Scheme

3.1 Recovery block reconfiguration

Whenever a try by a node with the primary try block, say A, results in an acceptance test failure, then the node attempts to process the current data with the other try block, say B. If that retry is successful, then the node faces a choice between picking B as the new primary try block and sticking to A as its primary try block. This will depend on among other things the role that the node had played before the acceptance test failure. In any case, this reconfiguration of the recovery block in a node that has experienced an acceptance test failure is a design parameter that may significantly affect the performance of the DRB implementation.

Three different strategies for recovery block reconfiguration are presented in Figure 2. Under strategy 1, the node that failed and has recovered through a retry, selects a new primary try block. That is, the try block used in the successful retry becomes the new primary try block. For example, if the primary node fails in its try with block A but recovers through a retry with B, then it starts playing the role of the backup node with block B as its new primary try block. Therefore, both the new primary and the new backup nodes use the same try block B as their new primary try blocks. Note also in Figure 2 that if the backup node fails in its try with its original try block B, it chooses block A as its new primary try block after recovery, thereby creating another situation where both nodes use the same try block, A, as their new primary try block. This is the drawback of strategy 1 because if certain input data that cannot be properly processed by the new primary try block arrive, then both nodes will fail, thereby leaving backward recovery as the only possible follow-on action.

Strategy 2 is aimed at always keeping the diversity of the primary try blocks used by the two nodes. Under the strategy, the primary node always uses try block A as its primary try block while the backup node always uses try block B as its primary try block. As shown in Figure 2, once the primary node fails in its try, the roles of the primary and backup nodes are reversed as well as the roles of the primary and backup try blocks in each node. On the other hand, if the backup node fails in its try, then the roles of the nodes and of the try blocks in each node remain unchanged. Therefore, the diversity of the primary try blocks is always maintained between the two nodes. A disadvantage of this strategy is that if try block A has a residual design error, it is theoretically possible to have a frequent exchange of roles between the two nodes. However, such frequent exchange will take place only if block A frequently fails to process input data properly. The probability of having such a primary try block in a software system produced through reasonably rigorous design and testing processes is considered very small.

Strategy 3 is another aimed at always keeping the diversity of the primary try blocks used by the two nodes. Under the strategy, a node that has failed, recovered through retry with a backup try block, and assumed the role of the backup node, sticks to the same primary try block used before the failure. Therefore, try block A is the permanent primary try block in the node assigned as the initial primary whether the node stays as the primary node or not. Similarly, try block B is the permanent primary try block in the node assigned as the initial backup. A disadvantage of this strategy is that try block B can play the role of the primary try block in the primary node even if try block A was produced as the preferred design.

Overall, strategy 2 is considered to be the most attractive among the three due to the reasons mentioned above. Strategy 1 is the least attractive. When there is no preference between the two try blocks based on the

quality of the computing services they provide, then strategy 3 should be equally attractive as strategy 2.

3.2 Cooperation between the partner nodes

The cooperation between the partner nodes is needed in order to facilitate fault detection and (computing station) recovery. The main design issue here is how often the nodes should interact. There is a trade-off between execution overhead due to interaction between partners and fault latency (the time elapsed from the occurrence of a fault to its detection). One important interaction point is where acceptance test results are exchanged. Figure 3 depicts a scenario of interaction where both nodes pass their acceptance tests. The actions marked Status-2 represent deposit of acceptance test results into buffer memories by both nodes. By action Check-1*, the primary node investigates if the backup node is still alive and well. If the primary node finds out that the backup node has not taken Status-2 actions during X recent data processing cycles, where X is a threshold value chosen by the system operator, then the primary concludes that the backup is dead. Regardless of its finding, the primary node proceeds to deliver its computation results to the successor computing station. By action Check-1, the successful backup node checks the acceptance test result of the primary node. If the acceptance test result of the primary node has not been deposited into the buffer, then the backup node waits until either it becomes available or a timeout event occurs. In the latter case, the backup concludes that the primary has died and proceeds to deliver its computation results to the successor computing station.

Another important interaction point is where the success of delivery of computation results to the successor computing station is confirmed. In Figure 3, the primary node delivers its computation results and then deposits a notice of its successful delivery into the buffer by action Status-3. Thereafter the primary node proceeds to enter the next data processing cycle. Meanwhile, by action Check-2 the backup node investigates if the primary has succeeded in delivery of computation results. It continues this investigation until either the notice is picked up or a timeout event occurs.

In both actions Check-1 and Check-2, it takes more time for the backup node to learn the death of the primary node via a timeout mechanism than to learn the action results of the alive primary via notices produced by the primary. The timeout values must be carefully chosen so as to avoid both excessive fault latency and false alarms.

Figure 4 depicts an interaction scenario where the primary node fails in its acceptance test while the backup node passes its acceptance test. The figure again illustrates cooperating actions Status-2, Check 1, Status 3, and Check-2.

Both Figure 3 and Figure 4 show another type of cooperating action Status-1. This is not so essential as the aforementioned actions taking place at the two interaction points. By action Status-1, each node can deposit a notice regarding its successful pickup of input data and successful establishment of a recovery point. This information is used later during action Check 1 (or Check 1*). It is useful for fast recognition of the death of the partner node in the case where the partner died between the completion of the previous data processing cycle and the beginning of the current data processing cycle. If a node finds during its action Check-1 that the partner has not taken action Status-1, then the checking node can conclude much before the occurrence of the normal timeout event that the partner is dead. Therefore, Status-1 is an example case of trading execution overhead for fault latency reduction.

The most complex case in operation of the DRB scheme that deserves consideration is when both nodes fail in their acceptance tests and recover through retries with their backup try blocks. See Figure 5. This situation can be created due to transient hardware faults but not by software faults. The reason is because if one of the try blocks, A and B, is incapable of properly handling the current data, the two nodes cannot both succeed in their retries (under any of the retry strategies discussed in preceding sections). The recovery effect produced by the DRB computing station in this case is that of backward recovery. In some applications, this may mean the loss of the current data processing cycle. The key question here is which node will assume the role of the primary after its successful retry and thus deliver its computation results to the successor computing station. Two of the reasonable strategies are discussed below.

One strategy is to make the initial primary node to resume the role of the primary after learning that the backup node also failed in its first try (via the notice deposited by the backup node through the first action Status-2). In other words, the initial primary node has a higher priority in resuming the role of the primary regardless of which node completes its successful retry first. Figure 5 shows such a case. Another strategy is to make the node that completes its successful retry first (i.e., the node that takes the second action Status-2 first) to assume the role of the new primary. In order to support this strategy, the inter-node communication mechanism used between the partner nodes must facilitate unambiguous ordering of the Status-2 actions taken by the two nodes. The latter strategy requires slightly more complex implementation but has an advantage of effecting faster recovery in the case where the retry by the initial primary node is completed long after completion of the retry by the initial backup.

3.3 Fault detection by the predecessor and successor computing stations

In order to support a computing station operating under the DRB scheme, its predecessor computing station attaches sequence number tags to the messages that contain computation results of the predecessor station before sending them to the DRB station. This message sequence number is used by the successor computing station in recognizing redundant messages that may be sent by the DRB station. More specifically, if the primary node in the DRB station crashes after sending a message containing its computation results to the successor station but before notifying its partner (backup) node of its successful delivery, then the partner will also send a message to the successor station. Therefore, the successor station will end up with data received from both partner nodes of the DRB station. The sequence numbers attached to the messages enable the successor station to recognize the redundancy in the two messages and discard the redundant message that has arrived later. The successor also realizes that the DRB station has detected a fault.

In the DCSs where a predecessor station can read the contents of the input buffers of its successor DRB station, the predecessor station can detect the failure of a

node in the DRB station by comparing the lengths of the message queues in the input buffers of the two partner nodes. If the queue lengths differ by more than Z, where Z is a threshold value selected by the system operator, then the predecessor identifies the oldest message in the longer message queue and records the sequence number attached to the oldest message together with the current time. The predecessor computing station checks the message queue again later when at least Y seconds have elapsed, where Y is another threshold value chosen by the system operator. If the predecessor finds the same oldest sequence number in the queue, then it concludes that the node owning the message queue is dead. Normally a node in the DRB station will detect the death of its partner node sooner than the predecessor computing station does. Therefore, in systems where it does not save much by making the predecessor send data to only one alive node of the DRB station, the fault detection capability of the predecessor station discussed here is not so important.

3.4 Connection between the computing stations incorporating the DRB scheme

When the successor computing station of a DRB station is also a DRB station, the messages containing computation results produced by the predecessor DRB station must of course be delivered into both input buffers (connected to the partner nodes) in the successor DRB station. In the case of a DCS in which the predecessor DRB station can deliver its computation results simultaneously to both partner nodes in the successor DRB station, there is no difference to the predecessor DRB station whether the successor station is a DRB station or not.

On the other hand, in a DCS in which the predecessor DRB station delivers its computation results sequentially to the two partner nodes in the successor DRB station, it is useful to slightly refine the protocol for cooperation between partner nodes in the predecessor DRB station. The area of refinement is the point of interaction where Status-3 and Check 2 actions in Figure 3 are taken. The Status-3 action needs to be divided into two steps: the first step is to generate a notice regarding successful delivery of computation results to the primary node in the successor DRB station whereas the second step is to generate a notice regarding successful delivery of computation results to the backup node. The Check-2 action needs to be expanded similarly in order to facilitate distinguishing the following two cases. Case 1) the sending node has successfully delivered its computation results to the primary node in the successor DRB station but crashed before delivering the results to the backup successor node; Case 2) the sending node crashed in the middle of delivering its results to the primary successor node. This discrimination of the two fault cases is important in minimizing the probability of delivering redundant results to the primary successor node.

4 Issues in Design of Non-disruptive Rejoin Capabilities

An effective way of prolonging the life of a DRB computing station is to provide the capability of repairing a failed node of the DRB station and rejoining the repaired node into the DRB station. The repair here may involve replacing the failed node with a spare node. The rejoin of the repaired node must be accomplished without disrupting the real time computing service of the DRB station. An extension of the DRB scheme that possesses such non disruptive rejoin capabilities is called a repairable DRB scheme in this paper. In order for a repaired node to rejoin the DRB station, it must obtain cooperation from both the partner node and the predecessor computing station.

The cooperation of the partner node is needed to bring the computation state or the local database of the rejoining node up-to-date. The type of database discussed here is typically resident in main memory. The simplest case is where the local database of the primary node can be copied in one atomic step to the rejoining backup node without impairing the real-time computing service of the primary node. This requires availability of sufficient slack time in the primary node between a certain consecutive pair of data processing cycles. In section 5 an experimental implementation of such a simple case is presented. This database duplication problem becomes highly complicated if the database cannot be copied in one atomic step. For example, after providing a copy of the first segment of the local database to the rejoining backup node, the primary node will proceed to process new input data and this data processing may involve updating the first segment of the local database. Therefore, when the primary node conducts the second copying step, it should provide not only a copy of the second segment of the local database but also information on the updating done on the first segment since the last copying step. This multi-step copying problem remains as a subject for future study.

The cooperation of the predecessor computing station is needed for two reasons. First, in order to function as a new backup node in the DRB station, the rejoining node must be recognized by the predecessor station and the latter should start sending data to the former. Secondly, the input buffer of the rejoining node must be made consistent with that of the primary node. Among many conceivable approaches to this input buffer consistency problem, a simplistic one is to have the predecessor station hold new data without delivering them into the input buffers until all the data in the input buffer of the primary node are processed and the local database of the node is copied to the rejoining node. Such an approach is feasible only in the systems in which single step copying of the local database is effective. Another approach is to have the predecessor station hold new data until the data in the input buffer of the primary node are copied to the input buffer of the rejoining node and provisions are made by the two partner nodes to accept new data without losing the feasibility of completing the database duplication in a reasonable amount of time. Therefore, both the input buffer consistency handling and the database duplication are parts of the state restoration process and thus they are intimately tied together.

5. Experimental Validation

In order to validate the logical soundness of a repairable DRB scheme and to evaluate the performance impacts of the implementation strategies discussed in preceding sections, an experimental implementation was carried out. The real time DCS testbed established in the authors' institution, called the Macro Dataflow Network (MDN), was used in this experimental implementation. The inter node connection approach adopted in the MDN is based on the use of a two port buffer memory as a medium for connecting a pair of microcomputers. The access time of the two port buffer memory developed in house is the same as that of the on board memory. A sufficient number of these buffer memory modules were con

structed to configure a variety of network topologies involving six microcomputer nodes.

A virtual machine called Extended Concurrent Pascal Machine (ECPM), a combination of a software nucleus and node hardware, was established on each node of the MDN to support distributed application processes communicating through monitors [Bri77]. A real-time application program running on the MDN was also developed. This program comprises several different modules distributed to run on the nodes of the MDN. A repairable DRB scheme was incorporated into a computing station running an important module of the application program. First, the logic acceptance test was designed to check if computation results fall within specified boundaries. This logic acceptance test was augmented with a watchdog timer that provided the time acceptance test. Detectors for exceptions such as arithmetic overflow, divide-by-zero, etc., were also incorporated to contribute to the acceptance test function. Secondly, two try blocks that differ mainly in arithmetic precision were designed. Try block A was designed to perform arithmetic of higher precision, but to have higher risks of running into overflow conditions than try block B. Also, the non-disruptive rejoin capability was incorporated with an implementation of a single-step database copying scheme.

Experiments were conducted to evaluate both the execution overhead caused by the introduction of the repairable DRB scheme and the recovery time. The amount of time spent for a data set to pass through a computing station was measured in both the case of a computing station not equipped with DRB mechanisms and the case of a DRB computing station. During experimentation, faults were injected to examine their impacts on system performance. The types of faults studied include: 1) total node failure; 2) transient hardware faults; and 3) software faults such as arithmetic overflow, etc. This experimental investigation produced a firm evidence of the fast recovery capability and the logical soundness of the repairable DRB scheme in the systems in which efficient inter-node connection media are used. The details of the implementation and measurement results can be found in [Kim87].

6. Summary

Various issues that arise in implementing the DRB scheme were discussed in this paper together with some promising approaches. Issues in extending the DRB scheme with non-disruptive rejoin capabilities were also discussed. The results of an experimental implementation of a repairable DRB scheme indicate firmly the fast forward recovery capability and the logical soundness of the scheme. The DRB scheme is aimed at preventing faults from crossing the station boundaries. For protecting against faults leaking through the guards established by the acceptance test, supplementary schemes are needed. In order to further improve the effectiveness and the practical utility of the DRB scheme, additional research is needed in areas such as multi-step database duplication, integration of the DRB scheme and other hardware fault tolerance mechanisms, and design of effective recovery blocks [And81,Avi84].

Acknowledgement: This work was supported in part by the National Science Foundation under Grant No. INT-8796259, in part by the Office of Naval Research under Contract No. N00014-87-K-0231, and in part by the US Army under Contract No. DASG60-85-C-0061.

References

[And81] Anderson, T. and Lee, P.A., 'Fault Tolerance: Principles and Practice', Ch.9, "Software Fault Tolerance", Prentice Hall Int'l, London, 1981.
[Avi84] Avizienis, A. and Kelley, J.P.J., "Fault Tolerance by Design Diversity: Concepts and Experiments", Computer, Vol.17, No.8, Aug. 1984, pp. 67-80.
[Bri77] Brinch Hansen, P., 'The Architecture of Concurrent Programs', Prentice Hall, NJ, 1977.
[Hec76] Hecht, H., "Fault-Tolerant Software for Real-time Applications", Comp. Surveys, Dec. 1976, pp. 391-407.
[Hor74] Horning, J.J., Lauer, H.C., Melliar-Smith, P.M., and Randell, B., "A Program Structure for Error Detection and Recovery," Lecture Notes in Comp. Sci., Vol.16, Springer-Verlag, New York, NY, 1974, pp. 171-187.
[Kim84] Kim, K.H., "Distributed Execution of Recovery Blocks: an Approach to Uniform Treatment of Hardware and Software Faults", Proc. 4th Int'l Conf. on Distributed Computing System, May 1984, pp.526-532.
[Kim87] Kim, K.H. and Yoon, J.C., "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme", Technical Report UCI-CEP-87-1, Computer Engineering Program, University of California, Irvine, November 1987.
[Kop85] Kopetz, H. and Merker, W., "The Architecture of MARS", Proc. FTCS-15, June 1985, pp. 274-279.
[McD82] McDonald, W.C. and Smith, R.W., "A flexible distributed testbed for real time applications", Computer, Vol.15, No.10, Oct. 1982, pp.25-39.
[Ran75] Randell, B., "System structure for software fault tolerance", IEEE Trans. Software Eng., Vol. SE-1, June 1975, pp.220-232.

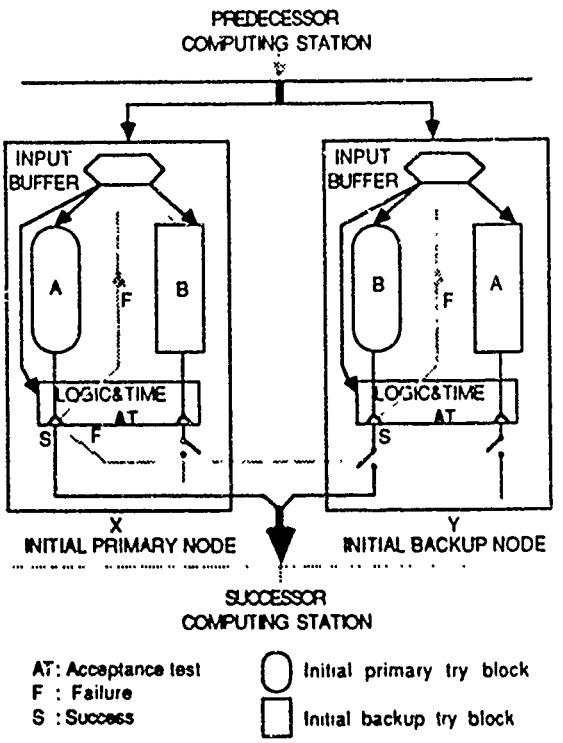


Figure 1. The basic structure of the distributed recovery block (DRB)

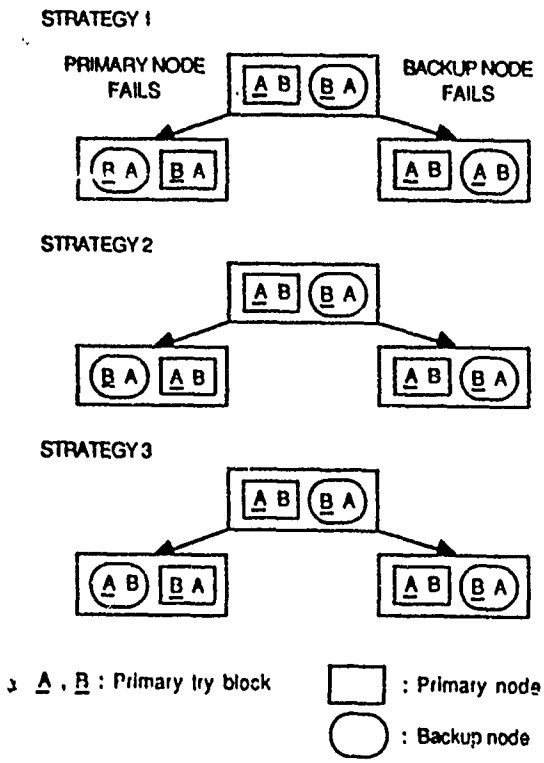


Figure 2. Strategies for recovery block reconfiguration

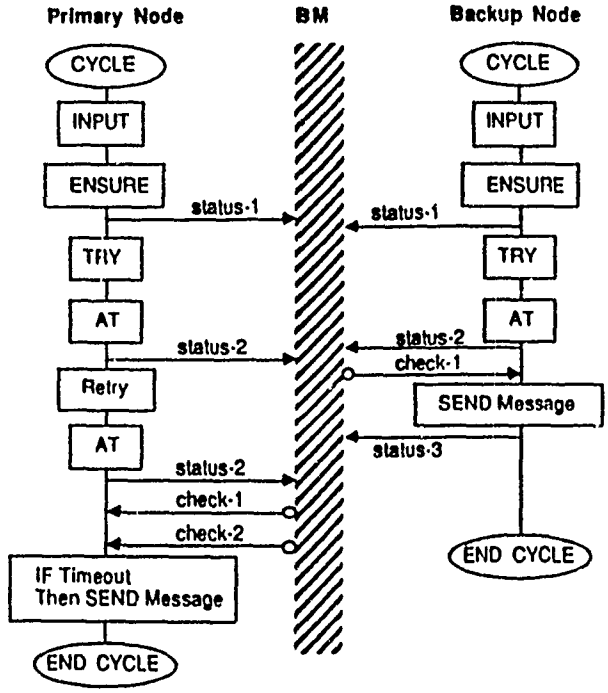


Figure 4. Cooperation scenario: Primary node fails in AT and backup node passes

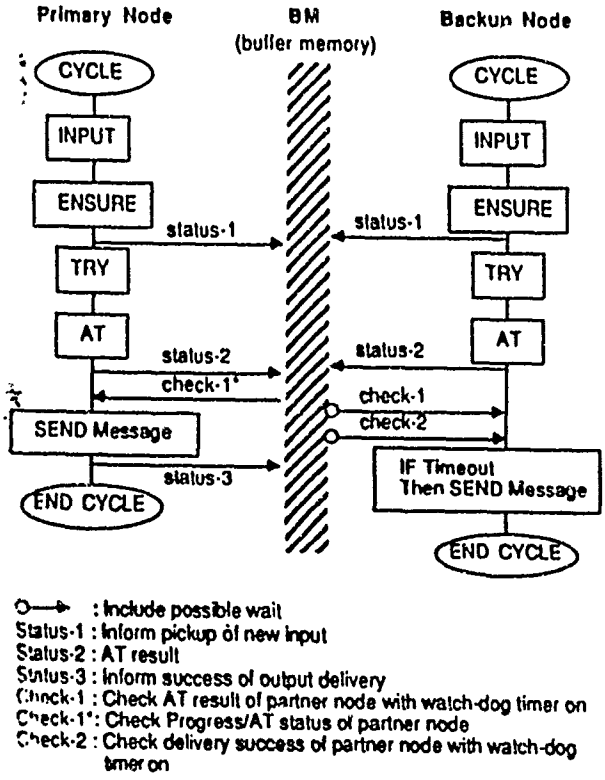


Figure 3. Cooperation scenario: both nodes pass

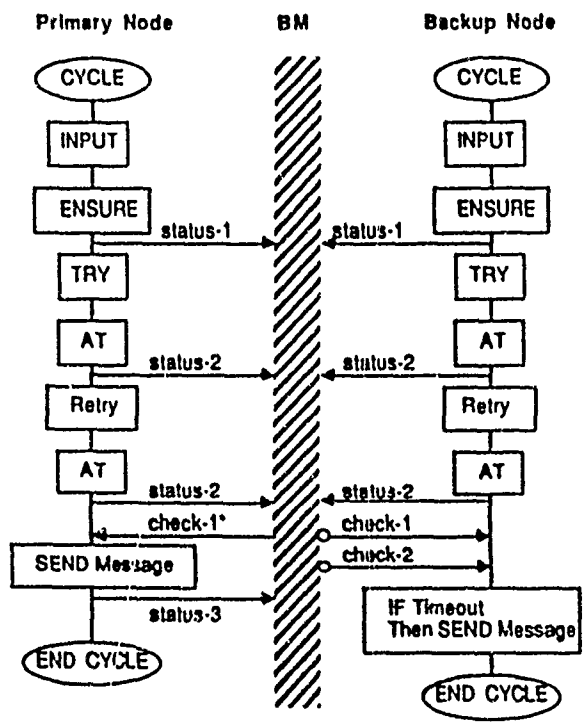


Figure 5. Cooperation scenario: both nodes fail in AT

Appendix A.IV

Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications

Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications

K. H. KIM, FELLOW, IEEE, AND HOWARD O. WELCH, MEMBER, IEEE

Abstract—This paper explores the concept of distributed execution of recovery blocks as an approach for uniform treatment of hardware and software faults. A useful characteristic of the approach is the relatively small time cost it requires. The approach is thus suitable for incorporation into real-time computer systems. A specific formulation of the approach that is aimed at minimizing the recovery time is first presented in this paper and it is called the distributed recovery block (DRB) scheme. The DRB scheme is capable of effecting forward recovery while handling both hardware and software faults in a uniform manner. An approach to incorporating the capability for distributed execution of recovery blocks into a load-sharing multiprocessing scheme is also discussed. Two experiments aimed at testing the execution efficiency of the schemes in real-time applications have been conducted on two different multimicrocomputer networks. The results clearly indicate the feasibility of achieving tolerance of hardware and software faults in a broad range of real-time computer systems by use of the schemes for distributed execution of recovery blocks.

Index Terms—Acceptance test, computing station, distributed recovery block, experimental validation, fault tolerance, forward recovery, load sharing, recovery time.

I. INTRODUCTION

AS COMPUTERS are increasingly used in critical real-time applications such as space navigation, air-traffic control, patient monitoring, power plant control, etc., users' demands for fault tolerance capabilities of such computers are also steadily increasing [1], [5], [6], [15] Such computer systems should be capable of tolerating failures of not only their hardware components but also their software components.

Historically the issue of software fault tolerance has received little attention from system designers in comparison to that paid to hardware fault tolerance. However, the software fault tolerance problem is becoming an increasingly serious problem as software becomes larger and more complex and given that complete validation of sizable software is not feasible. Designers of safety critical systems are seeking, with increasing frequency, approaches for tolerance of both hardware and software faults [10], [13].

In many of the fault tolerance schemes explored in the past, efficient recovery of the system from the detected fault depended upon the success in locating the extent of the fault with high precision [8]. However, distinguishing hardware faults from software faults in a real-time computer system has often been a problem troubling the system designers. For example, symptoms of transient hardware faults and those of software faults are often very similar. If their effects are detected when the result of a computing task is rejected by a run-time acceptance test or run-time assertion check, then the system may run a hardware diagnosis but it will not reveal any fault in the system hardware. The only way to tell whether such a rejection of the task result was due to a transient hardware fault or a software fault is to retry the task with the same software. If the result is rejected again, then it is reasonable to conclude that the software is faulty. Or if the result is accepted this time, then a reasonable conclusion is that a transient hardware fault occurred during the previous execution of the task.

Therefore, distinguishing hardware faults from software faults is costly even if it is logically feasible. Also, introducing different treatments for all possible different faults can quickly add up to an intolerable system complexity. Naturally, computer system designers have long sought techniques for unified treatment of both hardware and software faults.

The purpose of this paper is to show that distributed execution of recovery blocks, a combination of both distributed processing and recovery block concepts, is a promising approach for unified treatment of both hardware and software faults. The *recovery block* [7], [13] is a language construct supporting the incorporation of program redundancy into a fault tolerant program in a concise and easily readable form. The syntax of the recovery block is as follows: *ensure T by B1 else by B2 ··· else by Bn else error.*

In the above description, T denotes the *acceptance test*, $B1$ denotes the *primary try block*, and Bk, $2 \leq k \leq n$, denotes the *alternate try blocks* All the try blocks are designed to produce the same or similar computational results. The

Manuscript received May 20, 1985, revised September 2, 1988 and November 25, 1988 This work was supported in part by the U.S. Army under Contracts DASG60-83-C-0042 and DASG60-82-C-0019, in part by the NASA Langley Research Center under Grant NAG-1-470, in part by the Office of Naval Research under Contract N00014-87-K-0231, and in part by NASA JPL under Contract NAS7-918-RE 182,443.

K. H. Kim is with the Computer Engineering Program, Department of Electrical Engineering, University of California, Irvine, CA 92717.

H. O. Welch is with Optimization Technology, Inc., Auburn, AL 36831.

IEEE Log Number 8926780.

acceptance test is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A *try* (i.e., execution of a try block) is thus always followed by an acceptance test. In a sense, the recovery block is an enclosure of some recoverable activities of a single process.

In the next section, a specific scheme for distributed execution of recovery blocks termed the *distributed recovery block* (DRB) scheme is developed. A different scheme which is based on the incorporation of the capability for distributed execution of recovery blocks into a load-sharing multiprocessing scheme is discussed in Section III. To test the execution efficiency of the DRB scheme and the load-balancing recovery block execution scheme, two experimental implementations and measurements were done, one started at the University of South Florida (U.S.F.) and continued at the University of California, Irvine (U.C.I.) and the other at the U.S. Army Advanced Research Center (ARC), Huntsville, AL. The experiment results are discussed in Section IV. Section V is a summary.

II. The Distributed Recovery Block (DRB) Scheme

Throughout this paper, only two try blocks, i.e., the primary and the alternate, are used to illustrate distributed execution of a recovery block for the sake of simplicity. Also, the specification of a limit on the amount of execution time allowed for each try block is an integral part of the recovery block discussed in this paper [6]. The allowable time specification is used by a watch-dog timer to ensure timely completion of tries. A try not completed within the time limit is treated as a failure. Therefore, the acceptance test can be viewed as a combination of both the *logic acceptance test* and the *time acceptance test*.

The real-time computer systems considered in this paper are of the distributed system type with the following characteristics.

1) A computer system consists of multiple *computing stations*, each executing one and only one recovery block. In other words, the smallest unit of software redundancy used is a version of the full application program running on a computing station.

2) The result produced from a computing station may become an input to another computing station or to the application environment. For the sake of convenience in discussion, we regard the environment as a computing station, too. (This is also the view taken in the MARS project [11].) We can then say that the result from one computing station is always an input to another computing station and vice versa.

3) A computing station may consist of one or more computing nodes. Multiple computing nodes can be used either in a load-sharing mode or in a redundant processing mode.

4) A computing station usually maintains a database containing time-varying information related to the application environment.

In the rest of this section, the *distributed recovery block* (DRB) scheme is presented without consideration of a load sharing multinode computing station. Distributed execution of a recovery block in a load-sharing computing station is discussed in Section III. The DRB scheme is developed in three steps in the following.

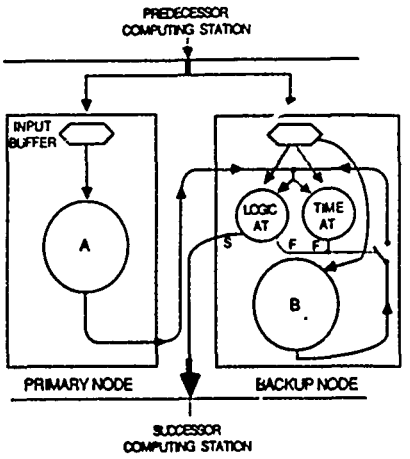


Fig. 1. Step 1: Distributed execution of try blocks.

1) *Distributed execution of try blocks:* Fig. 1 depicts the first step toward distributed execution of a recovery block. The primary node contains only the primary try block (A) whereas the alternate try block (B) and the logic and time acceptance tests run on the backup node. Therefore, the backup node is solely responsible for acceptance test and recovery. Both nodes receive the same data set simultaneously from the predecessor computing station. The term "data set" here refers to a set of data that is communicated between computing stations and activates an execution of a processing algorithm. The backup node turns its watch-dog timer on as soon as it receives a data set. Both nodes process the data set by using the try blocks contained within them. When the primary node has completed processing of the data set, the result is sent to the backup node for checkout by use of the logic acceptance test. If the result does not arrive at the backup node in time, then the result is said to fail the time acceptance test and the watch-dog timer in the backup node generates a "fault" signal. If the result arrives in time and passes the logic acceptance test, the result is used to update the database of the computing station and then forwarded to the successor computing station. On the other hand, if the result from the primary node fails the logic acceptance test or the time acceptance test, then the result from the alternate try block is checked out by the logic acceptance test. On passing the test, the result from the alternate try block is used to update the database and forwarded to the successor computing station.

Therefore, both the primary and alternate try blocks are executed concurrently. The failure of the primary node in producing an acceptable result can be caused by hardware faults, residual design errors in the primary try block, or both. The exact cause need not be identified for the recovery purpose. Whether the primary node fails due to hardware faults or due to the imperfect primary try block, the backup node takes the same recovery action.

2) *Replication of the logic acceptance test:* One serious problem in the scheme depicted in Fig. 1 is that if the backup node becomes inoperable, the entire computing station crashes because the decision making mechanism, i.e., the acceptance

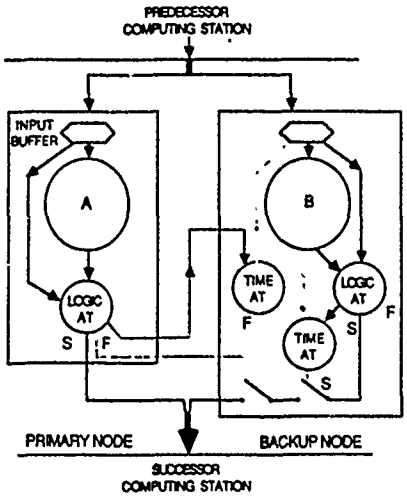


Fig. 2. Step 2: Replication of the logic acceptance test.

test, is in the backup node. Fig. 2 depicts the next step toward the DRB scheme, which is a result of an attempt to remove the major weakness of the scheme in Fig. 1. The logic acceptance test as well as the database of the computing station is now replicated in both the primary and backup nodes. Therefore, the result of each try block is checked out in the node in which the try block is running. If the primary node fails in its logic acceptance test, then it sends a notice to the backup node. The latter responds by forwarding its result generated from execution of the alternate try block to the successor computing station, provided that the result has passed both the logic and time acceptance tests and has been used to update the local copy of the database. Fig. 2 also shows two types of time acceptance tests provided by the backup node: a) one for ensuring timely completion of both the try block execution and the logic acceptance test by the primary node and b) the other for ensuring timely completion within the backup node itself.

In this scheme, the computing station does not crash even if the backup node becomes inoperable. One drawback of this scheme is that the primary node is abandoned when it fails to produce an acceptable result due to a residual design error in the primary try block. Therefore, a poor utilization of hardware resources can result and cause an unnecessary shortening of the lifetime of the computing station.

3) *Replication of the entire recovery block:* Fig. 3 depicts the DRB scheme which is based on replication of the entire recovery block and the database of the computing station into the primary and backup nodes. To be more specific, the DRB scheme is different from that depicted in Fig. 2 in two major ways. First, the complete acceptance test consisting of logic and time tests is now replicated in both the primary and backup nodes. Second, the try blocks are fully replicated in both nodes. However, the roles of the two try blocks are not the same in the two nodes. The primary node (say X) uses try block A as the primary try block initially, whereas the backup node (say Y) uses try block B as the initial primary. Therefore, until a fault is detected, both nodes receive the same input data, process them by use of two different try blocks (i e., block A on node X and block B on node Y), and check the results by use of the common acceptance test. As soon as each node passes the acceptance test, it updates its

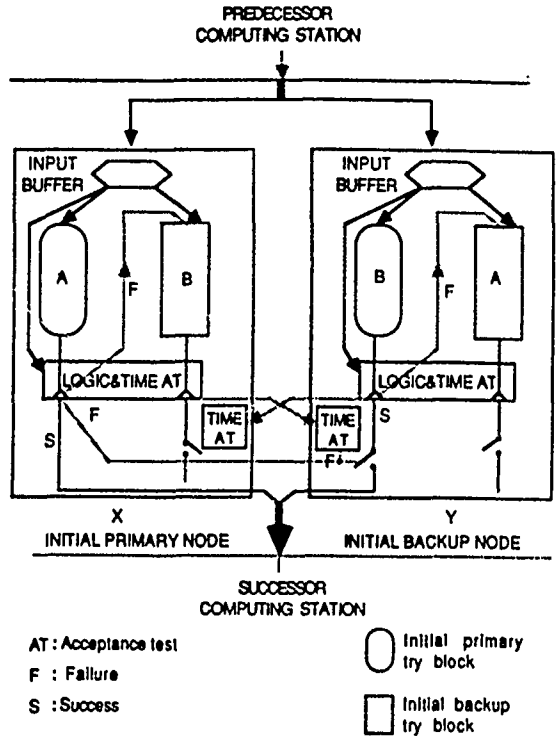


Fig. 3. The basic structure of the distributed recovery block (DRB).

local database. Both nodes perform all these tasks concurrently.

In a fault-free situation, both nodes will pass the acceptance test with the results computed with their primary try blocks. In such a case, the primary node notifies the backup of its success of the acceptance test. Thereafter, only the primary node sends its output to the successor computing station. However, if the primary node fails and the backup node passes its test, the backup node assumes the role of the primary, i.e., the nodes exchange their roles. To be more specific, the primary node attempts to inform the backup node upon its failure in passing the acceptance test. The backup node will take over the role of the primary as soon as it receives notice. If the primary node crashes completely, the backup node will recognize the failure of the primary upon expiration of the preset time limit. It will then become the new primary. On the other hand, if the backup node fails first, the primary node need not be disturbed.

Let us now examine in more detail the case where the primary node X fails and node Y becomes the new primary. If node X failed merely due to an error in the try block used (A), then the node attempts to become the backup node. The obvious goal is to be ready for a takeover and continued service in case the new primary node (Y) fails. The first step is to make a rollback-and-retry [4], [14] to process the data set that the node failed to process when it was the primary node. Processing of the data set is necessary to keep the application computation state of the node including the local database in the node up-to-date without disturbing the new primary node (Y). The retry should be done with the try block (B) other than that used during the previous failed try in order to maximize the probability of success.

If the next data set arrives before the new backup node

succeeds in the retry, then it must be queued in the input buffer; this means that the new backup node starts lagging behind the new primary node. In order for this scheme to work, the data arrival rate should be such that the new backup node may catch up with the primary node even if the former lags temporarily behind the latter. Whether the new backup node (X) stays with try block B for use as its primary try block after the successful retry with B or switches back to try block A, is a design choice.

In fact, selection of a primary try block is an important design choice not only for the new backup node but also for the new primary node. A strategy for this selection is called a *recovery block reconfiguration strategy*. Although many different strategies are conceivable, the strategies under which both the primary and the backup nodes use the same try block as their primary try block for a nontrivial period of time must be avoided. This is because a fault in the try block that causes a node to fail in processing a certain data set, will cause the other node to fail too. A conceptually simple and attractive strategy is the following:

> The current primary node always uses block A as the primary try block and block B as the backup try block whereas the current backup node always uses try blocks in the reverse order.

For example, suppose the primary node (X) (using A as the primary try block) fails in producing an acceptable result. First, the backup node (Y) (that has used B) attempts to deliver its result to the successor computing station and then the roles of the primary and backup nodes are reversed. For the new primary node (Y), block A must become the primary try block and thus the roles of the primary and backup try blocks are reversed. For the new backup node (X), block B must be used in a retry in order to bring the database in the node up-to-date. After the successful retry, block B remains as the primary in the new backup node. Therefore, under this reconfiguration strategy, a failure of the current primary node accompanies reversal of the roles of the nodes as well as reversal of the roles of the try blocks in "both" nodes. This strategy is attractive for two reasons. First, the two nodes always execute two different try blocks. An advantage here is that if a data set causes one of the try blocks to fail but not both of them, then one acceptable result can be sent to the successor computing station with little delay. Second, the current primary node always uses A as the primary try block and try block A is generally designed to produce better quality output than, if not the same quality as, try block B. One drawback of this strategy is that if try block A has a residual design error, it is possible to lead to frequent turnover of the roles between two nodes for some input data streams. However, the probability of such events occurring may be very small. Also, each occurrence of such failure caused by block A will be followed by a quick recovery.

As mentioned before, if the backup node (Y) fails first, then the primary node (X) need not be disturbed. The backup node will just make a retry with try block A to achieve localized recovery. If the backup node became totally inoperable, this arrangement is of course useless. However, if the failure was due to an error in try block B, then the localized recovery attempt can result in the backup node being available for continued service when the primary node fails later.

Under the DRB scheme, both hardware and software faults are treated in a uniform manner. Yet the scheme does not require any additional software design. It is just an efficient way of utilizing both hardware and software resources to maximally prolong the lifetime of a computing station. The recovery time is minimal since maximum concurrency is exploited in the redundant try block execution. If fact, the effect of forward recovery is achieved.

III. Distributed Execution of Recovery Blocks with Load Balancing

In the case where a computing station uses a large number, say n, of nodes for the purpose of load sharing, i.e., multiprocessing for the purpose of obtaining a throughput higher than what is obtainable with a single node, a straightforward application of the DRB scheme would be to group n computing nodes into $n/2$ primary-backup node pairs. Such an arrangement reduces the throughput potential to half of what is achievable with an irredundant operation of nodes. This is inevitable if the system application requires the fastest possible recovery from failures. However, in the applications where rollback-and-retry is an acceptable mode of recovery, the arrangement mentioned above is regarded as a wasteful way of using computing nodes. A more cost-effective approach is to connect the nodes loosely through queues containing data sets and allow each node to dynamically select its next task among the tasks of executing the primary try block, the alternate try block, and the acceptance test. Fig. 4 depicts such an approach. In this scheme, queues must be located in the global storage area that can be accessed by all the processing nodes of the computing station. The database of the computing station must also be located in the global storage area. Each node may select its next job from any of the four queues, input data queue (IDQ), try result queue (TRQ), arrival time record queue (ATRQ), and retry data queue (RDQ). The operations performed with the data set picked up from each of these queues are as follows.

Case 1) Input data queue (IDQ): A node that selected a data set from this queue executes the primary try block (A) with the data set and deposits the result (together with the original input data) into the try results queue (TRQ).

Case 2) Try results queue (TRQ): A node that selected this try result data set first removes a record in the arrival time record queue (ATRQ) which corresponds to the selected result data set. The processing of the ATRQ is discussed in Case 3. The node then executes the logic acceptance test. If the result is acceptable, an irrevocable update of the database of the computing station is carried out with the result. The result is then moved into the validated results queue (VRQ) for pickup by the successor computing station. If not acceptable, the data set is moved into the retry data queue (RDQ).

Case 3) Arrival time record queue (ATRQ). Each data set in this queue is an arrival time record containing both the arrival time and the maximum expected processing time of the corresponding data set in the input data queue (IDQ). This

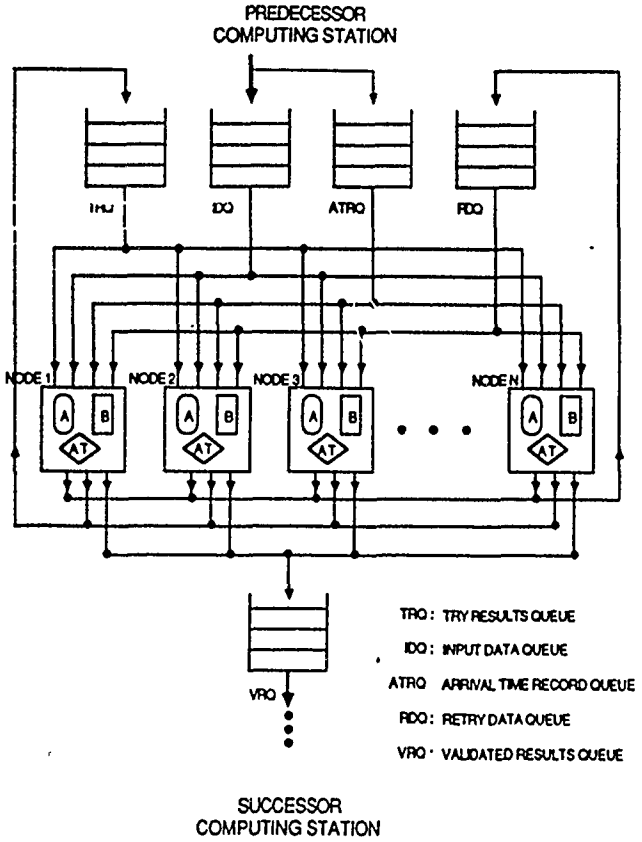


Fig. 4. A scheme for distributed execution of recovery blocks with load balancing.

information is used in determining if a timing fault has occurred or not. In other words, a node that selected an arrival time record checks if the record is too old. If so, the node moves the corresponding data set in the input data queue (IDQ) into the retry data queue (RDQ).

Case 4) Retry data queue (RDQ): A node that selected a data set from this queue executes the alternate try block (B) and deposits the results into the try results queue (TRQ).

As an illustration of the operation of this load-sharing multinode computing station, assume that a data set, say D, has just been produced by the predecessor computing station. The data set D is entered into IDQ and at the same time its arrival time record is entered into ATRQ. Now assume that node 3 has been idle and looking for work. When node 3 checks IDQ, it discovers data set D and picks it up. Node 3 then executes the primary try block A with data set D and deposits the processed result $D1$ into TRQ. Suppose that node 2 is idle at this time and it soon discovers the result data set $D1$ in TRQ. Node 2 then executes the acceptance test with both $D1$ and the original input data set D. Suppose that the result was a failure. Node 2 then moves D and the corresponding arrival time record into RDQ. Another idle node, say node 1, will soon discover D in RDQ, execute the alternate try block B with D, and deposit the result $D2$ into TRQ. Node 3 is idle at this time and it soon discovers $D2$ in TRQ. Node 3 thus executes the acceptance test with both $D2$ and D. Now the result is a success and node 3 updates the computing station database with $D2$. Node 3 also removes D (and the corresponding arrival time record) from RDQ and moves $D2$ into the validated results queue (VRQ) for pickup by the successor computing station.

One drawback of this scheme is that an "insane" node can disrupt a significant portion of the network, thereby causing a significant performance degradation. For example, it may either get stuck to a queue or repeatedly pick up new data sets and produce unacceptable results. Or it may continue to pick up data sets from the try results queue (TRQ) and reject them even if they are good. However, it may be possible to implement the scheme in such a way that the probability of an "insane" node causing a significant disturbance becomes very small. This is a subject worthy of further study. It may also be worthwhile to explore some other possible ways of combining the DRB scheme and load balancing schemes.

IV. Experimental Validation

In this section, two experiments carried out to test the time overhead of the schemes for distributed execution of recovery blocks are described. One experiment was carried out initially at the University of South Florida (U.S.F.) and later at the University of California, Irvine (U.C.I.) and the other at the U.S. Army Advanced Research Center (ARC), Huntsville, AL.

The performance of a scheme for distributed execution of recovery blocks is a function of several factors. For example, the type of communication medium used to connect computing nodes is a major factor in determining the performance. The higher the communication bandwidth is, the better the performance becomes. The networks used in both experiments are tightly coupled ones. Another example of a factor that impacts the performance is the programming language used in implementing the scheme.

A. Experiment Conducted at U.S.F. and U.C.I. with the DRB Scheme

The network configuration used is depicted in Fig. 5. This network facility has been named the Macro-Dataflow Network (MDN). A node used here is a Z8000-based single-board microcomputer called the OEM-Z8000. The connection medium used between nodes is a two-port buffer memory developed in-house and consisting of two independent memory banks of 16K bytes each. Nodes can exchange data through a two-port buffer memory nearly at the rate of local (on-board) memory access. A software nucleus implemented on each node supports concurrent programs consisting of asynchronous processes communicating through monitors, in particular, the programs written in the Extended Concurrent Pascal language [3], [9].

The distributed application program executed on the MDN during the experiment was written in Extended Concurrent Pascal [3], [9]. The distributed functions of this program are indicated in Fig. 5. The DRB scheme was incorporated into nodes 3 and 6 (performing the Analyzer 2 function). Node 5 (data generator) simulates a real time device which generates stimulus data sent to the rest of the network and accepts the response (command). The remaining five data processing nodes contain an *input* (i.e., stimulus data) *classification process*, various *analysis processes*, constituting the intelli-

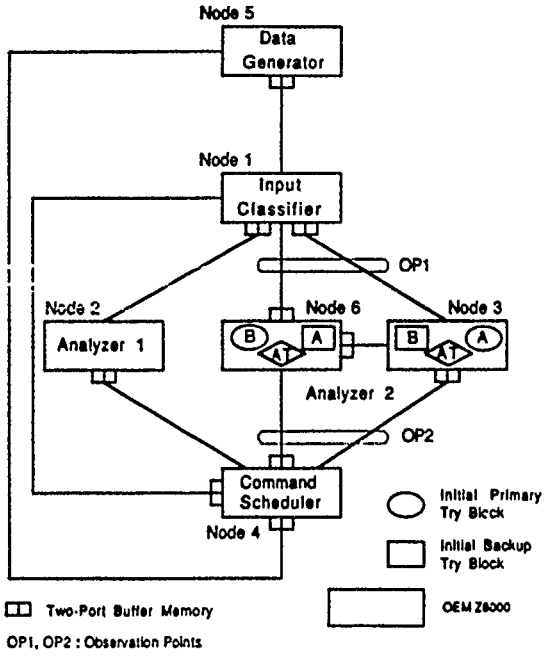


Fig. 5. The network configuration used for experimentation of the DRB scheme.

gence of the solution algorithm, and a *control command scheduler process* that delivers the network's response to the real-time device. The stimuli data from node 5 are first handled by the input classification process which distributes inputs to the rest of the network. The command scheduler process honors requests from various analysis processes to schedule commands for the real-time device.

The "travel times" of data sets passing through a computing station were measured to determine the execution overhead caused by the introduction of the DRB scheme into the network. As a part of facilitating this measurement, "observation points" were established in the network. When a data set arrives at the designated observation point in the network, the node stamps the real-time and saves a copy of the time-stamped data in its local memory. When enough measured data are obtained, the time-stamped data are transferred to another computer system for data analysis. The observation points are usually established at the points where the nodes are ready to send messages to the successor nodes and also at points where the nodes have received messages.

In this experiment, two observation points were set up in the network. Fig. 5 shows these points established in the network. Observation point 1 (OP1) is set up where the primary and backup nodes have taken the data set from the queue buffers connected to the predecessor node. Observation point 2 (OP2) is set up where both nodes are ready to put the result data sets into the queue buffers connected to the successor node.

During experimentation, faults were injected to examine their impacts on system performance. The types of faults studied include. 1) total node failure (simulated by node reset), 2) transient hardware faults simulated by random changes in the contents of certain memory locations, and 3) software faults such as infinite looping, arithmetic overflow, etc. The DRB incorporated into nodes 3 and 6 in Fig. 5 was written in Extended Concurrent Pascal and executed on an OEM-Z8000 microcomputer with a clock rate of 4 MHz.

The DRB overhead consists of interprocess communication among nodes and the execution of the acceptance test. Fig. 6 shows such overhead for incorporating the DRB scheme into the network. One curve represents the delay between OP1 and OP2 in the case of using the DRB whereas the other curve represents the delay in the case without the DRB. The gap between the two curves is the execution time increase due to the incorporation of the DRB. The average execution time increase is approximately 30 ms. Moreover, the upper curve in Fig. 6 also shows instances (marked by X's) where arithmetic overflows occur in the primary node and the fast recovery capabilities of the DRB scheme are exercised. We noted that the fault occurrences and subsequent recovery actions did not cause any visible degradation of the system performance. In the absence of fault, the execution time increase is caused mainly by the execution of the acceptance test and the communication of the acceptance test success to the backup node.

Considering the inefficient implementation language (Extended Concurrent Pascal), and the slow processor (4 MHz Z8001) used, the amount of execution time increase shown in Fig. 6 is at least 20 times higher than that expected in the systems built with current off-the-shelf hardware and software tools. For example, use of a processor running at 20 MHz will result in speedup by a factor of 5. Use of a more efficient language (an Assembly language in the extreme case) will result in additional speedup by a factor of about 4.

Fig. 7 shows the case where the primary node is reset, resulting in the crash of the node. Later an arithmetic overflow occurs in the remaining node. The recovery from the first fault (the crash of the primary node) took about 60 ms. This recovery time is largely a function of the timeout period used in the DRB. When the second fault (the arithmetic overflow) occurred in the remaining node after the crash of the first node, the node had no choice but to roll back and retry with try block *B*. Therefore, the recovery time was very high, i.e., about 290 ms, as shown in the figure. Again, the recovery time can be easily reduced by a factor of 20 by implementing real application systems with current off-the-shelf tools.

The data discussed above indicate that the DRB scheme can be used in many real-time applications with tolerable amounts of time overhead.

B. Experiment Conducted at ARC with the Scheme for Distributed Execution of Recovery Blocks with Load Balancing

The primary objectives of the experiment at ARC were as follows:

1) to establish the initial feasibility of real-time recovery from hardware failures, failures of an algorithm to compute reasonable results, and failures of task completions,

2) to measure the impact of the recovery mechanisms on processing resource utilization and critical response times, and

3) to demonstrate that the approach of distributed execution of recovery blocks can be applied to real time application processes and distributed operating systems.

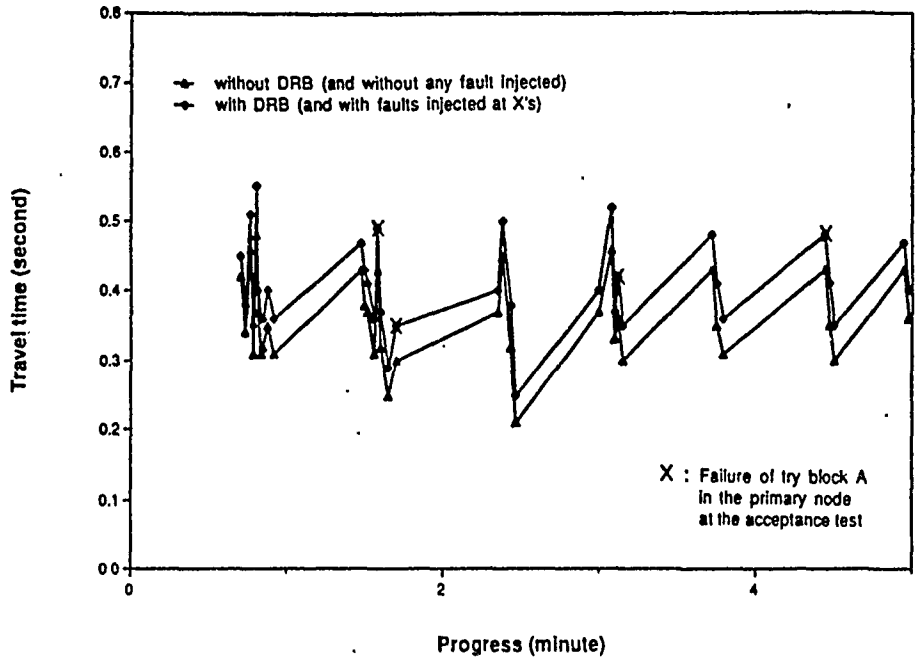


Fig. 6. Data travel time measured.

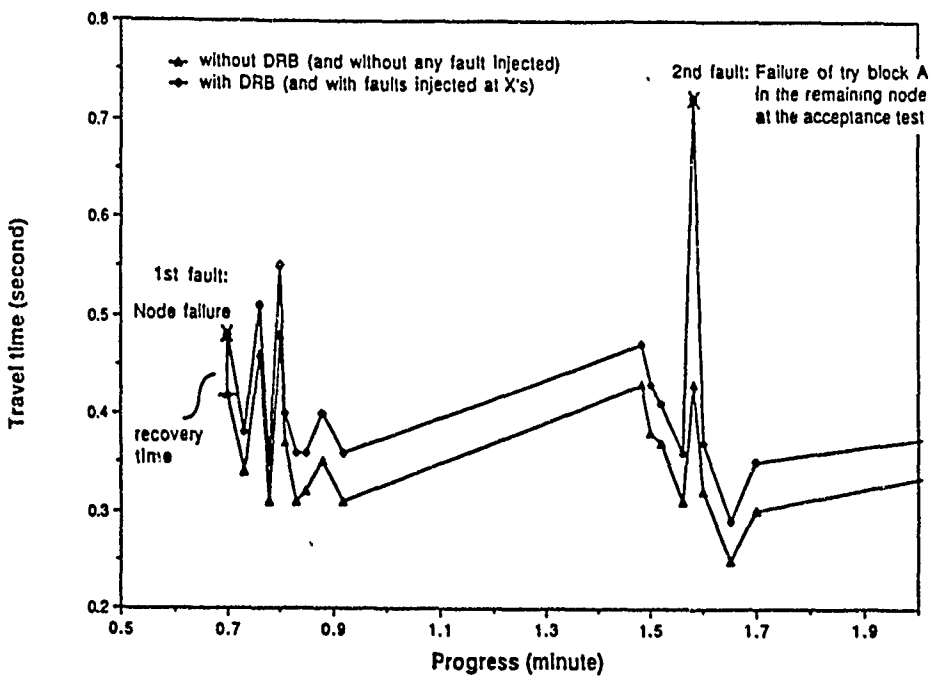


Fig. 7. Data travel time measured.

Unlike the experiment conducted on the MDN, this experiment dealt with a load-sharing multinode computing station. The recovery time requirements imposed on the multinode station are such that the load-balancing recovery block execution scheme discussed in Section III can be used to meet the requirements.

Fig. 8 depicts the distributed real-time system adopted as the baseline configuration for this experimental research. The system is a distributed implementation of a closed loop radar control system. The hardware base of this system is the Crossbar Multi-Microcomputer System (CMS) [12]. The CMS is composed of six computers (OEM-Z8000) interconnected to 12 shared memory modules through a crossbar switch. The fully parallel crossbar switch transfers the normal Z8001 memory access signals (address, data, control) directly to the shared memory. The queues shown in Fig. 8 are kept in the shared memory. The software was written in PDL [16] (an extension of Pascal supporting concurrent programming) One processor is dedicated to simulation of a radar and assimilation of data returned from the radar (RRA), four processors to a set of three analysis processes ($A1$, $A2$, $A3$), and a sixth processor to the radar scheduling process A shared database maintained in the shared memory contains data on the objects tracked by the radar.

The application program scheduler (APP scheduler) in each of the four processors schedules analysis processes for execution. It polls the three input data queues (A1Q, A2Q, A3Q), which contain radar return data sets, in a round robin

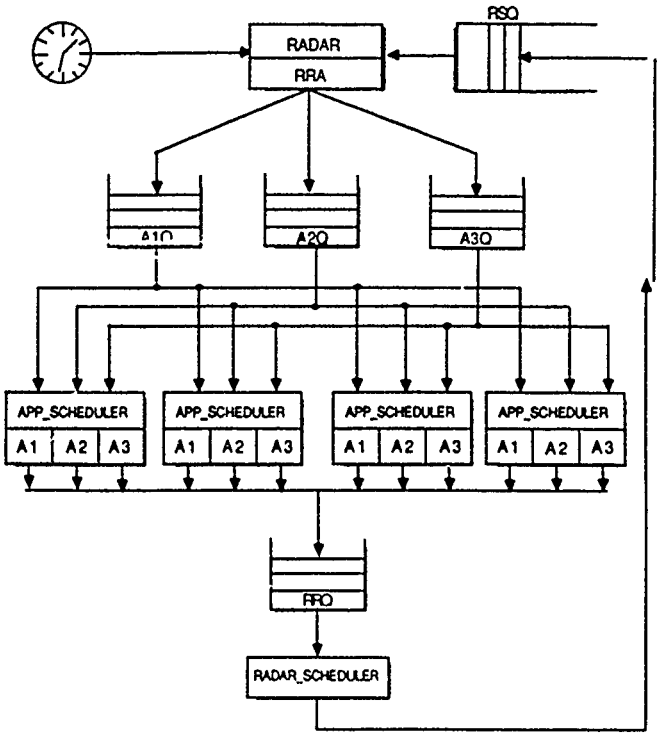


Fig. 8. Baseline network configuration for radar control.

fashion looking for work. When an entry is found, the scheduler activates an appropriate analysis process to work on the data set. After processing the data set, the analysis process places a request for a tracking pulse in the radar request queue (RRQ) connected to the radar scheduler (RS). The radar scheduler honors such requests by placing them in appropriate slots within the radar schedule queue (RSQ) connected to the radar.

The system calendar clock is accessible to all processors as a time base. Every processor in the CMS is an OEM-Z8000 capable of performing about 0.350 MIPS (million instructions per second) and regarded as a simulator of a target processor which should be much more powerful. Therefore, the radar control simulation is run at a down-scaled rate. The scale factor is based on the ratio of the target machine instruction execution rate to the OEM-Z8000 instruction execution rate. This time scaling approach was adopted to enable reasonably accurate evaluation of the time cost of an experimental technology such as the fault tolerance scheme studied here in an application context very close to the real operating environment.

Fig. 9 depicts the fault-tolerant distributed system configuration with the load-balancing recovery block execution scheme incorporated. The recovery block was incorporated only for analysis process $A3$. The process AT determines if the result computed by $A3$ is within reasonable bounds based on flight dynamics. BA3 is a backup, independently coded analysis process.

All data sets produced by the radar return assimilator (RRA) and other processes are kept in the shared database. In fact, the data sets in the database never really enter any of the queues shown in Fig. 9, only the pointers to the data sets enter the queues. For example, the RRA process places a pointer to a data set into A3Q. If an APP scheduler picks the pointer and activates the analysis process $A3$, then $A3$ makes a copy of the data set pointed to by the pointer for its processing and places its result into TRQ. Later, a certain APP scheduler picks this result from TRQ and executes the acceptance test (AT). If the result is acceptable, an update of the database with the accepted result follows. If the result is not acceptable, the AT places a pointer to the original (unprocessed) data set into RDQ, thereby causing the backup analysis process BA3 to process the data set again. Also, when a pointer to a data set is originally entered into A3Q, a deadline for the analysis process $A3$ to process the data set is placed in ATRQ. ATRQ is processed by APP schedulers to detect data sets for which the processing by $A3$ is overdue. A relatively loose deadline of 30 ms was used in this experiment to ensure that no false alarms (reporting fault detection when there are no faults) would occur.

This fault-tolerant network configuration (Fig. 9) was compared to the baseline network configuration (Fig. 8) by running them against the same radar load and measuring a number of performance parameters. Node failures were injected several times during the analysis $A3$, but each time a successful recovery was accomplished.

Track response time is the time from the entry of a data set into A3Q to the insertion of a corresponding radar request into RSQ by the radar scheduling process. It was used as a measure of real-time computer system effectiveness in this study.

Fig. 10 is a track response time histogram. The track response times are increased by inclusion of the recovery block because the task sequence in the fault-tolerant configuration includes the acceptance test (AT) and because of the lengthened polling loop of the APP scheduler. The mean track response time rises from 1.8 to 2.6 ms which is still considerably below the allowable maximum, 40 ms, for this particular application. As mentioned earlier, these numbers represent the performance expected in the real systems built with the tools and components required by the applications.

Fig. 11 is a plot of maximum track response times for the data sets processed by $A1$, $A2$, and $A3$ in the fault-tolerant network configuration. The figure shows the maximum track response times over every 50 ms interval during the experiment. The large spike of 32 ms of $A3$ was caused by recovery from an injected processor failure in computing node. For false alarm control the timeout value for ATRQ was set at 30 ms. Seven additional small spikes are visible for injected algorithm errors detected by the acceptance test function.

Fig. 12 plots the aggregate computing node utilization for the four nodes assigned to analysis processes and the acceptance test for both the baseline configuration case and the fault-tolerant configuration case. It shows consistently greater node utilization for the fault-tolerant configuration and this is natural because incorporation of a recovery block increases the workload.

Fig. 13 shows the total node time spent for the acceptance test function expressed as a percentage of the total (four nodes) resource (computing time) available. The node time spent for the acceptance test never exceeded 8 percent of the total resource during the experiment. While this figure may vary with the logical complexity of the acceptance test, the results

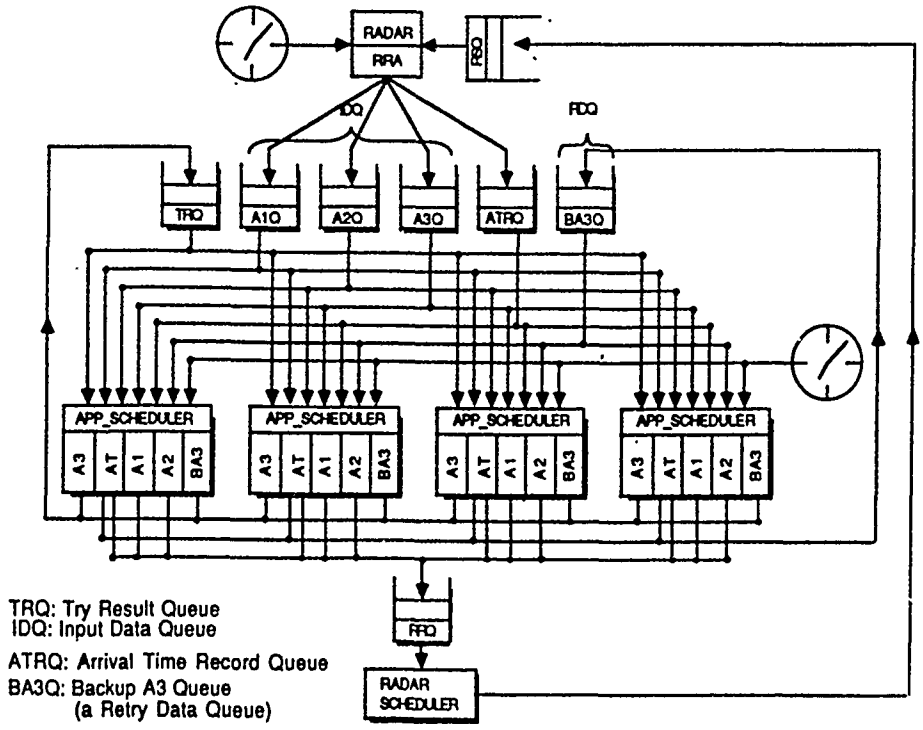


Fig. 9. Fault-tolerant network configuration with the load-balancing recovery block execution scheme incorporated.

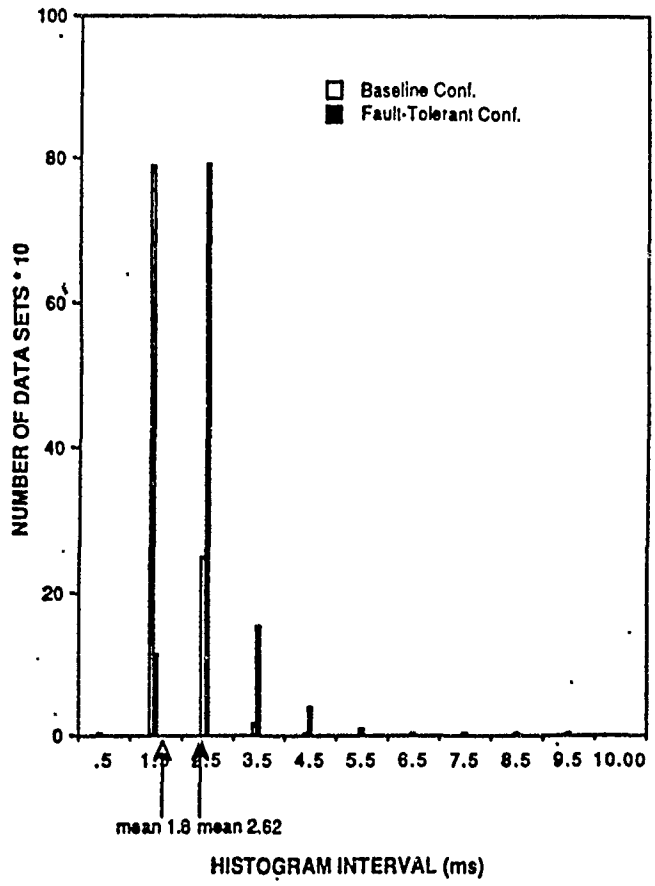


Fig. 10. Track response time histogram.

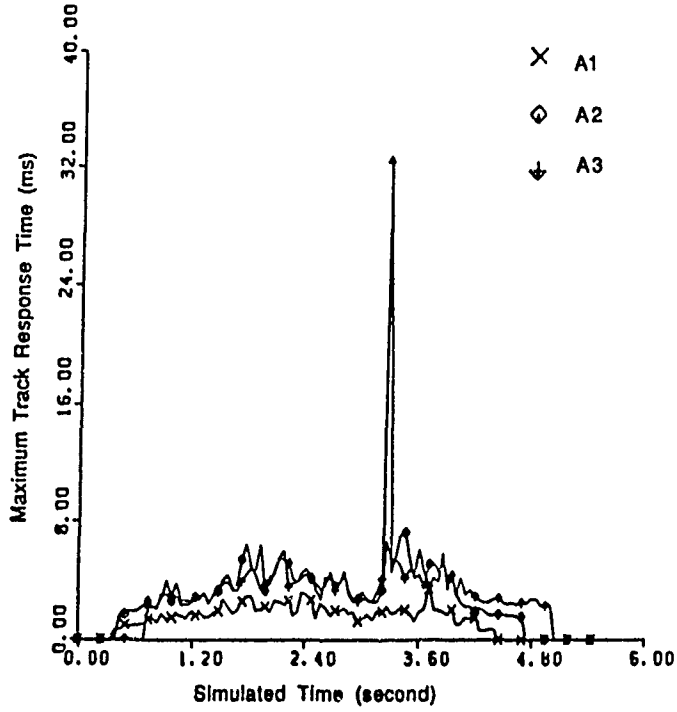


Fig. 11. Maximum track response times (fault-tolerant network configuration in Fig. 9).

of this experiment indicate that the acceptance test in radar control applications does not use excessive node resources. The reasonableness test adopted involved a linear extrapolation of the old states of an object to a new state, a calculation of the error vector, and chi squared test.

Fig. 14 shows that total node time spent for the backup $A3$ function as a percentage of the total (four nodes) resource available. Eight faults including both software faults and node crashes were induced stochastically to simulate the detection by both the acceptance test and the APP scheduler and to cause the execution of the backup $A3$ function. These faults show as small spikes of node utilization in Fig. 14. The larger spike at approximately 2.6 s into the simulation is due to two induced algorithm errors in the same 50 ms data collection interval and

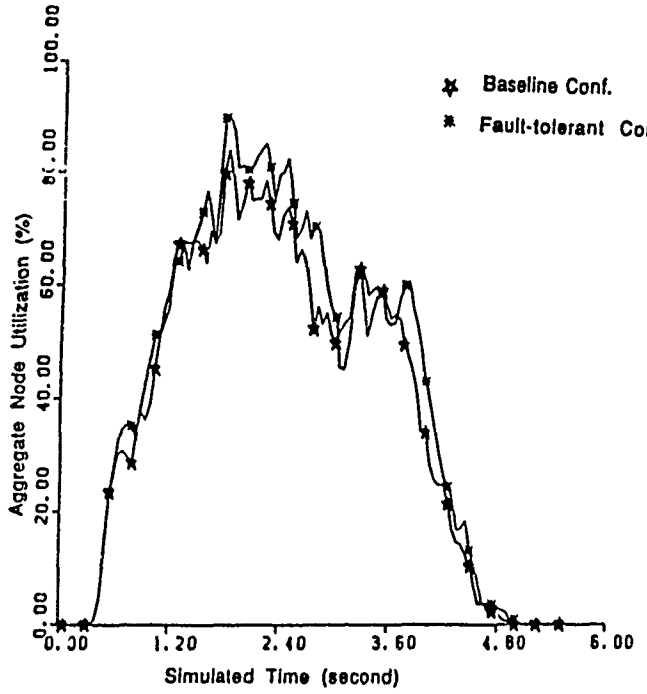


Fig. 12. Aggregate node utilization.

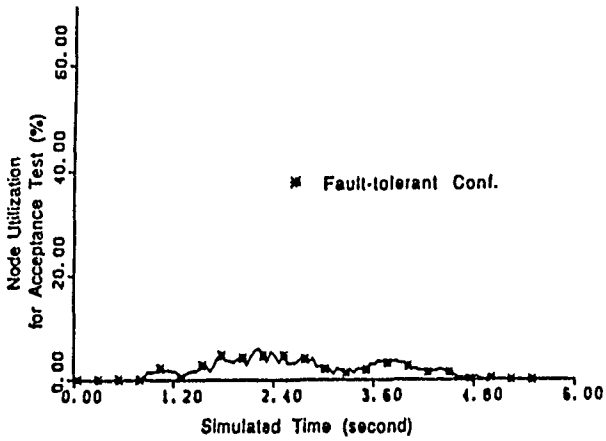


Fig. 13. Node utilization for acceptance test.

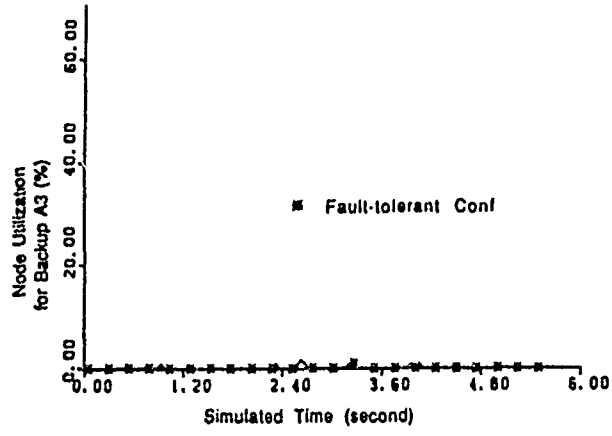


Fig. 14. Node utilization for backup $A3$.

two resulting executions of the backup $A3$. As shown in the figure, the backup $A3$ processing is not a significant workload for this system.

The absolute values of the numbers reported are not so much significant as the fact that the results of this experiment indicate the feasibility of using the load-balancing recovery block execution scheme in many real-time applications which impose stringent response time requirements.

V. Summary

The concept of distributed execution of recovery blocks was presented in this paper. Two practical fault-tolerant distributed computing schemes based on the concept were formulated and experimented with. They are the results of a search for techniques for uniform treatment of hardware and software faults in real-time computer systems. The experimental results, although limited, are encouraging in that they match with our expectation on the small time costs of the schemes.

The cases of nested recovery blocks should in principle pose no serious problems since enlosing recovery blocks and nested recovery blocks can be assigned to different computing stations. Suppose recovery block RB2 is nested within a try block A of the enclosing recovery block RB1. The computing station executing RB2 under the DRB scheme interfaces with the computing node executing try block A. The latter may treat the former as if it were treating a peripheral.

Some of the basic questions related to the effective use of recovery blocks for software fault tolerance, e.g., design of effective acceptance tests, correlation among the faults of multiple try blocks [2], etc., still remain unsatisfactorily answered and not helped by the distributed execution schemes. Until those questions are settled, the recovery block approach to software fault tolerance remains an immature technology.

Acknowledgment

The authors wish to express gratitude to C. G. Davis, E. C. Foudriat, R. Grafton, N. Goddard, H. Hecht, C. Holland, V. Kobler, W. C. McDonald, J. Rohr, T. D. Smith, D. Thomas, and A. van Tilborg for their encouragement during the course of this work. The first author also wishes to acknowledge the valuable assistance received from the colleagues in the DREAM (Distributed Real-Time Ever Available Microcomputing) Laboratory (previously at U.S.F. and currently at U.C.I.). The second author performed the work reported here while at Unisys Corporation, Huntsville, AL, and wishes to acknowledge the support of W. Moquin and the Unisys Crossbar Microprocessor project staff.

References

[1] T. Anderson and B. Randell, Eds., *Computing System Reliability*. Cambridge, England: Cambridge University Press, 1979.

[2] A. Avizienis, M. R. Lyu, and W. Schutz, "In search of effective diversity: A six-language study of fault-tolerant flight control software," in *Proc. IEEE Comput. Soc. 18th Int. Symp. Fault-Tolerant Comput.*, June 1988, pp. 15-22.

[3] P. Brinch Hansen, *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

[4] K. N. Chandy and C. V. Ramamoorthy, "Rollback and recovery strategies for computer programs," *IEEE Trans. Comput.*, pp. 59-65, June 1972.

[5] C. G. Davis and R. L. Couch, "Ballistic missile defense: A supercomputer challenge," *IEEE Computer*, pp. 37-46, Nov. 1980.
[6] H. Hecht, "Fault-tolerant software for real-time applications," *Comput. Surveys*, pp. 391-407, Dec. 1976.
[7] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," in *Lecture Notes in Comput. Sci.*, vol. 16. New York: Springer-Verlag, 1974, pp. 171-187.
[8] K. H. Kim, H. Hecht, J. Huang, and M. Naghibzadeh, "Strategies for structured and fault-tolerant design of recovery programs," in *Proc. IEEE Comput. Soc. Int. Comput. Software Appl. Conf. (COMPSAC)*, Nov. 1978, pp. 651-656.
[9] K. H. Kim, "Evolution of a virtual machine supporting fault-tolerant distributed processes at a research laboratory," in *Proc. IEEE Comput. Soc. 1st Int. Conf. Data Eng.*, Los Angeles, CA, Apr. 1984, pp. 620-628.
[10] K. H. Kim, "Software fault tolerance," in *Handbook of Software Engineering*. C. R. Vick and C. V. Ramamoorthy, Eds. New York: Van Nostrand Reinhold, 1984, ch. 20.
[11] H. Kopetz and W. Merker, "The architecture of MARS," in *Proc. IEEE Comput. Soc. 15th Int. Symp. Fault-Tolerant Comput.*, June 1985, pp. 274-279.
[12] W. C. McDonald and R. W. Smith, "A flexible distributed testbed for real time applications," *IEEE Computer*, vol. 15, pp. 25-39, Oct. 1982.
[13] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, pp. 220-232, June 1975.
[14] J. A. Rohr, "STAREX self-repair routines: Software recovery in the JPL-STAR computer," in *Dig. IEEE Comput. Soc. Int. Symp. Fault-Tolerant Comput.*, 1973, pp. 11-16.
[15] O. Serlin, "Fault-tolerant systems in commercial applications," *IEEE Computer*, pp. 19-30, Aug. 1984.
[16] N. A. Vosbury, "The process design system," in *Proc. IEEE Comput. Soc. Int. Comput. Software Appl. Conf. (COMPSAC)*, 1979, pp. 374-379.

K. H. Kim (S'73-M'75-SM'86-F'89) received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1969, the M.A. degree in computer science from the University of Texas, Austin, in 1972, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1974.

From 1969 to 1971 he served as an officer in the Korean Army. From 1975 to 1986 he served on the faculty of the University of South Florida, Tampa, the State University of New York, Binghamton, and the University of Southern California, Los Angeles. While teaching at Binghamton, he also served as acting chairman of the Department of Computer Science for nine months. He is currently a Professor of Computer Engineering in the Department of Electrical Engineering at the University of California, Irvine. His current research interests are in the areas of reliable distributed processing and real-time software engineering. He is currently conducting both analytic and experimental research in the DREAM (Distributed Real-Time Ever Available Microcomputing) Laboratory.

Dr. Kim is a member of the Association for Computing Machinery and the IFIP Working Group 10.4, and a fellow of the IEEE Computer Society. He served as the Chairman of the IEEE Computer Society's Technical Committee on Distributed Processing from September 1984 to December 1986. In 1989 he plans to host the IEEE Computer Society's 9th International Conference on Distributed Computing Systems as the General Chairman.

Howard O. Welch (M'86) received the B.S. degree from Washington State University, Pullman, in 1961.

Since 1986, he has acted as Manager of Operations and Technical Director for Optimization Technology, Inc., Auburn, AL responsible for research and development contracts in software fault tolerance, computer testbed development, and parallel processing. He was previously with UNISYS Corporation, Huntsville, AL, active in parallel processing and software engineering research at the U.S. Army Strategic Defense Command Advanced Research Center.

Appendix A.V

Performance Impacts of Look-Ahead Execution in the Conversation Scheme

Performance Impacts of Look-Ahead Execution in the Conversation Scheme

K. H. KIM, FELLOW, IEEE, AND SEUNG M. YANG, MEMBER, IEEE

Abstract—**One of the basic issues that should be resolved in order to broaden the application range of the conversation scheme for design and execution of fault-tolerant concurrent programs is the control of the execution overhead inherent in the scheme. Recently, it has become known that under practical circumstances, the performance of a fault-tolerant multiprocessor/multicomputer system operating under the basic conversation execution scheme would be significantly affected by the synchronization required of the processes in exiting from a conversation. The look-ahead execution approach, that allows early finishing participant processes to exit from a conversation before other participants finish their conversation activities, is adopted here as a fundamental approach to reducing the synchronization overhead. Queueing network models are developed for both the system operating under the basic conversation execution scheme and the system operating under the execution scheme extended with the look-ahead capability. Based on the models, various performance indicators such as the system throughput, the average number of processors idling inside a conversation due to the synchronization required, and the average time spent in a conversation, are evaluated numerically for different application environments. The performances under the look-ahead execution scheme are compared against those under the basic conversation execution scheme. The results provide insights into the extent of benefits that can be brought in by the look-ahead execution approach.**

Index Terms—**Backward recovery, conversation, cooperating processes, fault tolerance, look-ahead, queueing network.**

I. INTRODUCTION

As the use of distributed computer systems in safety-critical applications has steadily increased in recent years, the design of fault handling capabilities into the concurrent programs running on the distributed hardware has become a subject of ...ious interest to system designers [1], [3], [4], [6], [12], [14], [18] The conversation scheme proposed by Randell in [15] is on. of the fundamental approaches to structured design of such fault-tolerant concurrent programs As discussed in [17] the scheme provides a means of facilitating failure atomicity and backward recovery in cooperating process systems in a manner analogous to that of the atomic action mechanism in object-based systems On the basis of the abstract notion of the conversation in [15], several concrete structuring approaches and supporting tools have been developed as described in [2], [5], [8], [9], and [13]. However, their utilities have not been fully tested and not much is known about the performance characteristics of the conversation scheme.

One of the costs of using the conversation scheme is the execution time increase due to the tight synchronization imposed among participant processes and due to the execution of the conversation acceptance test. This is an overhead. Another cost of interest to system designers is the recovery time, i.e., the time spent for recovering from detected faults. These time costs were analyzed in [10] by use of a queueing network model. The results showed among other things that under practical circumstances the system performance is significantly affected by the synchronization required of the processes in exit from a conversation, not by the probability of acceptance test failure.

A fundamental approach to reducing the synchronization overhead is the look-ahead. That is, by allowing the participant processes which complete their conversation activities including acceptance tests earlier than other participants to exit from the conversation and continue processing rather than to wait until all the participants have passed their acceptance tests, substantial reduction of the synchronization overhead can be achieved. When the participants exit from a conversation via a look-ahead, they should maintain the recovery points established on their entries to the conversation. This is because other participants may later fail in their acceptance tests in which case all the participants must be brought back to the recovery line (i.e., the entry points of the conversation) for retry. Therefore, the recovery costs may increase when the look-ahead is used. The possibility of incorporating the look-ahead capability into the conversation scheme was discussed in [7], [16], and [19]. In this paper, we analyze the impacts of the look-ahead approach on the performance of the conversation scheme.

Our interest here is in studying the inherent overhead, i.e., the overhead due to acceptance test and synchronized exit, not the scheduling overhead due to limitations in the available processors. Therefore, we consider the cases of using multiprocessor systems or tightly coupled networks of computers in which each processor is dedicated to running a single process This actually matches with the current trend in use of multimicrocomputer systems in real-time applications. The queueing network model developed in [10] for the case of the basic conversation scheme without the look ahead capability is extended in this paper to cover the cases with the look-ahead capability. One attractive feature of this queueing-network-

Manuscript received October 19, 1988; revised March 10, 1989 This work was supported in part by the NASA JPL and the U.S. Army SDC under Contract NAS7-918-RE-182443, in part by the Office of Naval Research under Contract N00014-87-K-0231 and Contract N0014-88-K-0622, in part by the University of California MICRO program and AT&T under Grant 88-123, and in part by the U.S. Air Force RADC.

K. H. Kim is with the Computer Engineering Program, Department of Electrical Engineering, University of California, Irvine, CA 92717

S M Yang is with the Department of Computer Science Engineering, University of Texas at Arlington, Arlington, TX 76019

IEEE Log Number 8928525.

based analytic evaluation is the feasibility of covering a broad range of situations. The specific performance indicators analyzed include the system throughput, the average number of processors idling inside a conversation due to the synchronization, and the average time spent in a conversation. Comparison of the performance in the two cases, i.e., the case of synchronous exit without look-ahead and the case of using look-ahead. reveals that the look-ahead approach offers potential for significant reduction of the execution overhead of the conversation scheme.

In the next section, a brief review of the basic conversation scheme is given together with an introduction of the look-ahead approach. Section III then discusses the execution environment considered in this paper and the queueing network models developed for both the basic conversation scheme and the scheme extended with the look-ahead. The models are used in Section IV to evaluate and compare the system performance under different execution approaches and workloads. Section V is the summary section.

II. Basic Conversation Structure and Look-Ahead

The conversation is a two-dimensional enclosure of recoverable activities of multiple interacting processes, in short, recoverable interacting session [8], [15]. As depicted in Fig. 1 it creates a "boundary" which process interactions may not cross. The boundary of a conversation consists of a *recovery line*, a *test line*, and the walls defining the membership. Each participant process contains one or more try blocks designed to produce the same or similar computation results as well as an acceptance test which is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A recovery line is a coordinated set of the recovery points of interacting processes that are established (possibly at different times) before interactions begin. A test line is a correlated set of test points at which computation results of interacting processes are checked out via execution of acceptance tests. The correlated set of acceptance tests used at a test line may be viewed as a single global acceptance called a *conversation acceptance test*. A conversation is successful only if all the interacting processes pass their acceptance tests at the test line. If any of the acceptance tests fails due to a residual design error in the try block used, a hardware malfunction, a timeout enforced by a watchdog timer, etc., all the processes roll back to the recovery line and retry with their alternate try blocks. These alternate try blocks collectively define an *alternate interacting session* (AIS) [and may be viewed as an *alternate interaction block* (AIB)], whereas the set of primary try blocks executed first after the processes enter the conversation define the *primary interacting session* (PIS) [and may be viewed as the *primary interaction block* (PIB)].

A process which is inside a conversation cannot interact with a process which is not in the convesation. Conversations must be strictly nested in two dimensions. That is, when conversation C nest is nested within conversation C, the set of processes that participate in nested conversation C.nest must be a subset of the processes that participate in C, the entire recovery line of C nest must be established after the entire

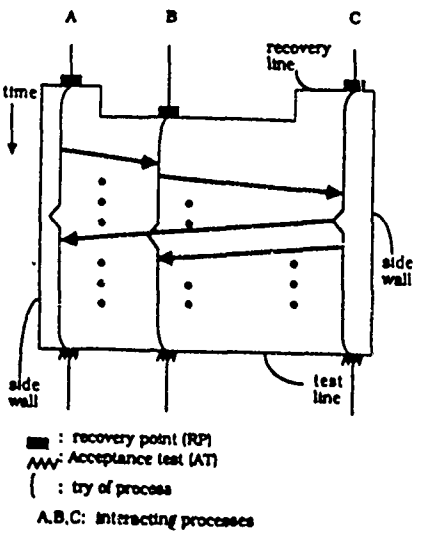


Fig. 1. Abstract conversation structure.

recovery line of C, and the entire test line of C.nest must be set before the entire test line of C.

In the basic conversation scheme sketched above, processes enter a conversation asynchronously but synchronize themselves before exiting from it. The synchronization considered here is of a special kind specifically required by the conversation scheme and thus different from the application-dependent synchronization required between cooperating processes. As mentioned in the preceding section, the synchronization can add significantly to the time cost of the conversation scheme. The look-ahead is a fundamental way of reducing this synchronization overhead. Under the look-ahead approach, each participant process leaves the conversation as soon as it passes its own acceptance test. It does so with the awareness of the possibility that another participant may execute an acceptance test later and fail in the acceptance test, thus making it necessary for the former to roll back to the recovery line of the conversation. Therefore, the look-ahead approach here is an optimistic approach and is aimed at trading increase in recovery costs for reduction of synchronization overhead.

A process that has exited from a conversation $C1$ via look-ahead may enter another conversation $C2$. If some participants of $C1$ are not participants of $C2$, then it is possible that $C2$ activities including acceptance tests are completed while $C1$ remains unfinished. In such a case, $C2$ should be treated as an unfinished conversation until $C1$ becomes completed. This is because if $C1$ fails, then all $C2$ activities that have taken place must be nullified as a part of the rollback to the recovery line of $C1$.

Although it is logically feasible to make provisions for a participant process to look ahead of the unfinished conversation to an unlimited extent, it is useful to limit the extent of the look-ahead with respect to controlling the implementation complexity. In most practical applications, there are natural limits to the extents of look-ahead possible, when a process progressing past the test lines of one or more unfinished conversations reaches a point where it needs to interact with other slowly following processes, it has to wait for the other processes to become ready for interaction. In addition, the

look-ahead should not be allowed to go past the points where critical irreversible actions, e.g., certain output actions, are taken. In the next section, the cases where the look-ahead is allowed to limited extents are considered.

III. The Execution Environment Assumed and Queueing Network Models

The characteristics of the hardware and the software of the systems considered are described first. Queueing network models are then developed in Section III-C.

A. Hardware Characteristics

The type of system considered in this paper is depicted in Fig. 2. A system consists of multiple, say N, computing nodes. Each node is equipped with a processor, a local memory, and a communication interface. The internode connection medium is either a high-speed bus or a common memory shared among multiple nodes. We assume that internode signaling is accomplished via interrupts or single-byte messages. Consequently, the communication overhead is negligible.

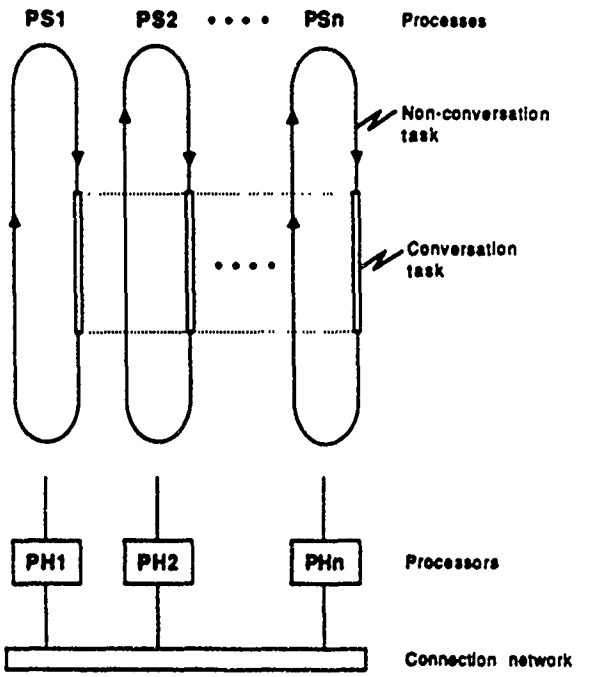


Fig. 2. A model of the fault-tolerant systems in parallel execution of conversations.

B. Software Characteristics

As shown in Fig. 2, each node in the system runs a single process which is a nonterminating cyclic program in execution. Each process cycle consists of two steps: a process step inside a conversation called a *conversation task*, and a process step outside a conversation called a *nonconversation task*. Each process alternates between two tasks and in every cycle all the processes participate in the same conversation. Therefore, we are dealing with systems in parallel execution of conversations.

A conversation is successful only if all the participant processes pass their acceptance tests. If any of the acceptance tests fails, all the processes roll back to the recovery line and retry with their alternate try blocks. It is assumed that the conversation construct contains one primary interaction block and an infinite number of alternate interaction blocks. This means that every conversation succeeds eventually.

If a process fails its acceptance test which is a part of conversation CONV, it broadcasts the failure message to other participant processes. Upon receiving a failure message, the processes which have been executing their conversation tasks belonging to CONV abandon the tasks (including acceptance tests) and join the group of failed processes. The processes which have already completed their conversation tasks belonging to CONV also join the group of failed processes. On the other hand, the processes which have not entered CONV and have been executing their nonconversation tasks will complete the tasks and then immediately after entering CONV, they will also join the group of failed processes After all the participants join the failed group, they retry with their alternate try blocks.

In this modeling study, the conversations nested within other conversations were not explicitly dealt with for the sake of simplicity. The following is a summary of the software characteristics assumed in this paper.

1) Each process alternates between its conversation task and nonconversation task.

2) All processes participate in the same conversation in every cycle.

3) Each process has an unlimited number of alternate try blocks.

4) Once a process fails its acceptance test, all other processes that have not finished the conversation tasks abandon the conversation tasks.

5) There are no conversations nested within other conversations.

The look-ahead capability is incorporated into the system in Fig. 2 with the scope of look-ahead limited. The *look-ahead scope* is defined here in terms of the active conversations that a process can participate in and leave from via look-ahead. In the simplest case, a process is allowed to make a "single conversation look-ahead," which means that a process can continue look-ahead as long as it has not exited from more than one unfinished conversation. In Fig. 3, for example, assume that process B completes its conversation task ($B2$) in CONV1 while process A is still executing CONV1. Process B can leave CONV1 because there are no other active conversations which it exited from. Assume further that process B completes its task ($B4$) in conversation CONV2 while process A is still executing CONV1. Now process B cannot leave CONV2 because CONV1 has not been completely validated. Therefore, under the single conversation look-ahead rule a process is not allowed to leave a conversation if there is an earlier initiated but unfinished conversation that the process has participated in.

Similarly, the two-conversation look-ahead rule can be defined. In this case, a process is not allowed to leave a

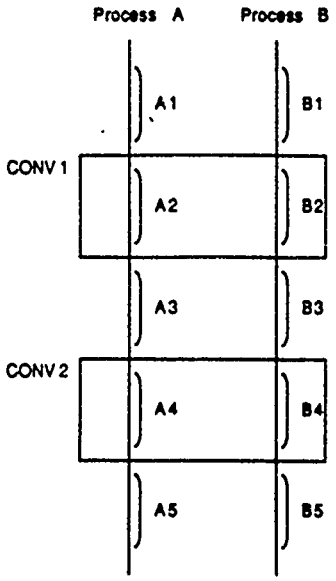


Fig. 3. An example of conversation structure.

conversation if the conversation is currently the third unfinished conversation that the process has participated in.

For convenience, the basic conversation scheme without the look-ahead capability is denoted by CL-0 in the rest of this paper whereas the conversation scheme operating under the single-conversation look-ahead rule and the scheme under the two-conversation look-ahead rule are denoted by CL-1 and CL-2, respectively.

C. Queueing Network Models

1) Model for CL-0 (Zero Look-Ahead): A queueing network model of the system shown in Fig. 2 operating under CL-0 is depicted in Fig. 4. Servers in the model represent processors and customers represent processes which alternate between two different tasks.

Each process moves among three different states: executing a conversation task, waiting for other processes to complete the conversation tasks, and executing a nonconversation task. In Fig. 4, the customers in $Q1$ represent the processes in execution of conversation tasks whereas the customers in $Q3$ represent the processes in execution of nonconversation tasks. The customers in $Q2$ represent the processes which have finished their conversation tasks and are waiting for others to finish. As soon as $Q2$ becomes full, all the processes in the queue move to either $Q3$ or $Q1i$, $1 \le i < \infty$, depending upon whether any of the processes has failed in its acceptance test or not. If they all have passed their acceptance tests, then they enter $Q3$. Otherwise, they all enter $Q1i$, $1 \le i < \infty$, the queue for their next alternate interacting session. Since $Q3$ is empty during the retry, there is no loss of information caused by merging all $Q1i$, $1 \le i < \infty$, into $Q1$. Fig. 5 depicts a more abstract representation of the queueing network model depicted in Fig 4. Therefore, N processes, where N is the total number of processes in the system, are distributed over three different queues.

Since each process runs on a dedicated processor, no waiting time is needed for the process to execute its task. This characteristic is correctly represented in the queueing network

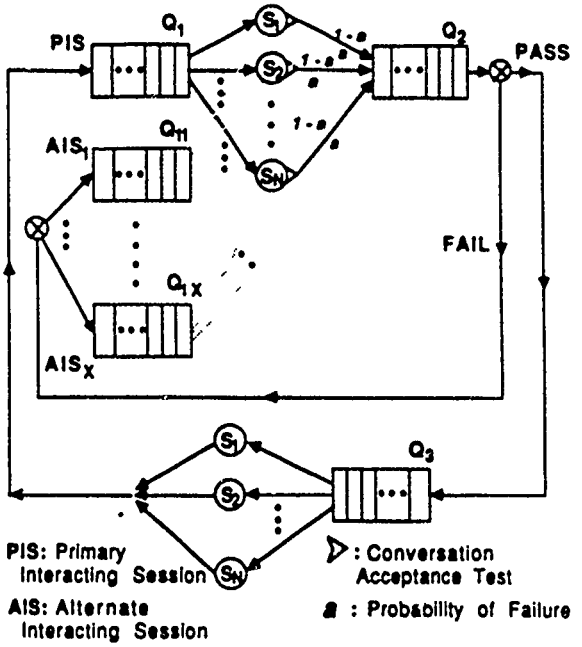


Fig. 4. A queueing network model of the system in Fig. 2.

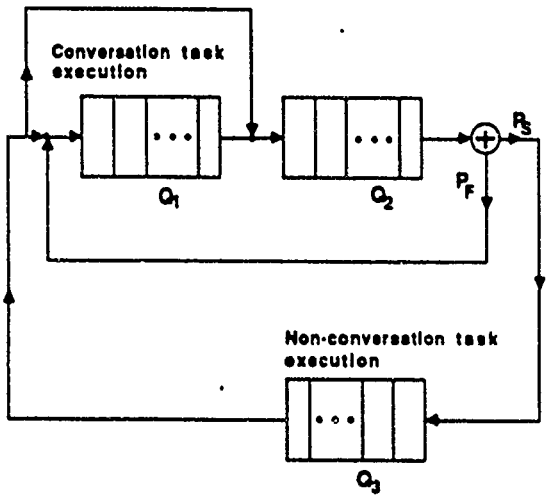


Fig. 5. An abstract representation of the queueing network model in Fig. 4.

model by the number of $Q1$ servers as well as the number of $Q3$ servers being always equal to N. Therefore, the execution time of a conversation task is represented by the length of stay in $Q1$, whereas the execution time of a nonconversation task is represented by the length of stay in $Q3$.

Let $(n1, n2, n3)$ represent a state of the queueing network in which $n1$ processes are in $Q1$, $n2$ processes are in $Q2$, and $n3$ processes are in $Q3$, where $N = n1 + n2 + n3$. $P(n1, n2, n3)$ denotes the stationary probability of the state. However, this does not represent all possible states since $(n1, n2, n3)$ does not indicate anything about whether any of the process in $Q2$ has failed its acceptance test or not. If any of the processes in $Q1$ fails its acceptance test, then all others in $Q1$ should immediately abandon their conversation tasks and enter $Q2$. From then on, a process which has been in $Q3$ proceeeds directly to enter $Q2$ without executing its conversation task in $Q1$ as it leaves $Q3$ and enters the conversation. This is shown as a bypass around $Q1$ in Fig. 5. When $Q2$ becomes full, all

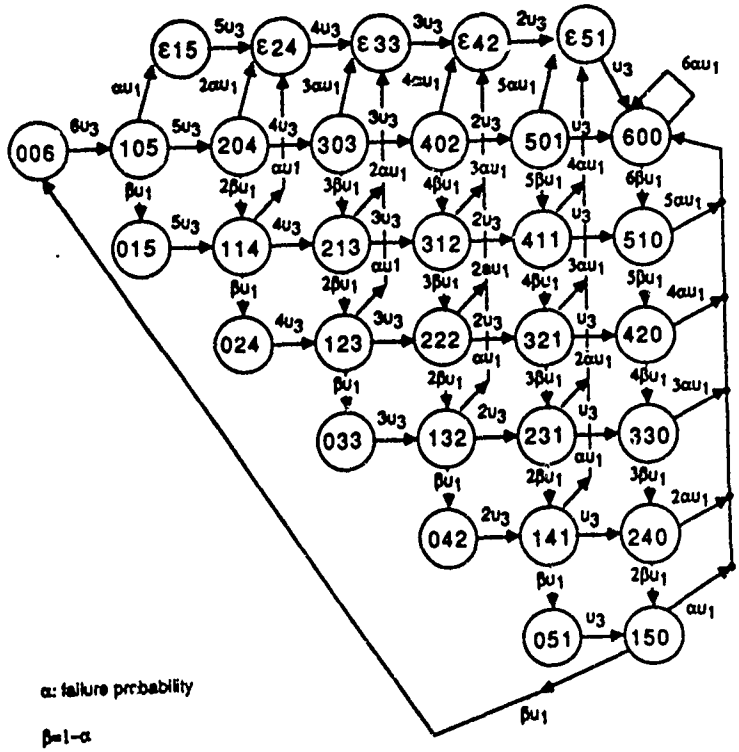


Fig. 6. State transition diagram for the six-process system operating under CL-0.

the processes return to $Q1$. Let $(\epsilon, n2, n3)$ represent an error state in which $n2$ processes are in $Q2$, $n3$ processes are in $Q3$, where $N = n2 + n3$, and at least one of the processes in $Q2$ has failed its acceptance test. $P(\epsilon, n2, n3)$ denotes the stationary probability of the state. As discussed above, when the network is in an error state $(\epsilon, n2, n3)$, $Q1$ is empty. The following assumptions are also an integral part of the queueing network model.

A1: The execution times of the conversation tasks and those of the nonconversation tasks are exponentially distributed with means $1/u1$ and $1/u3$, respectively.

A2: The probability of failure in an acceptance test for each process is α.

The transitions of the six-process system among all possible states are depicted in Fig. 6.

2) Model for CL-1 (One-Conversation Look-Ahead): The queueing network model developed above for the system operating under CL-0 can be extended as depicted in Fig. 7 to represent the system operating under CL-1. This queueing model contains two sets of three queues, i.e., queue sets ($Q1$, $Q2$, $Q3$) and ($Q1'$, $Q2'$, $Q3'$). The three queues in each set correspond to the three queues in Fig. 5. The customers in $Q1$ and $Q1'$ represent the processes in execution of conversation tasks whereas the customers in $Q3$ and $Q3'$ represent the processes in execution of nonconversation tasks. The customers in $Q2$ and $Q2'$ represent the processes which have finished their conversation tasks and are waiting for others to finish.

Therefore, this new queueing model looks similar to the model of the system in which processes alternate between two types of conversations. The only difference is that there are

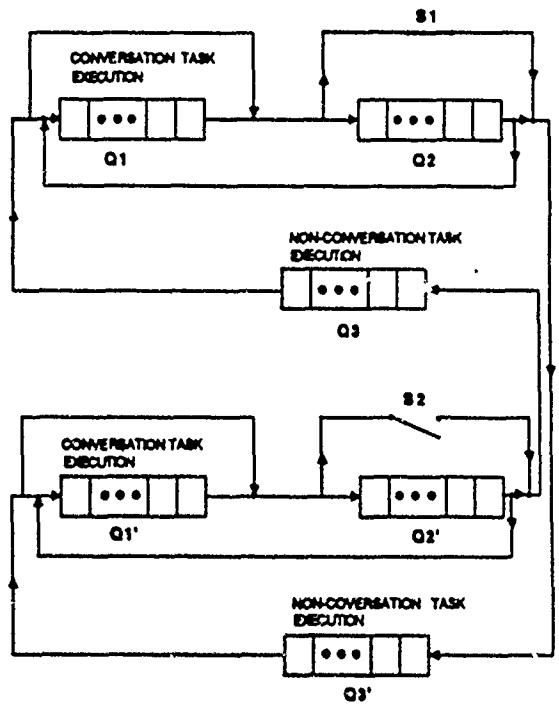


Fig. 7. A queueing network model for CL-1.

bypasses around $Q2$ and $Q2'$ with on-off switches, $S1$ and $S2$, respectively. In a sense, the bypasses are look-ahead paths and the switches enable/disable look-ahead. The look-ahead switches are opened and closed alternatively, i.e., if $S1$ is open, $S2$ is closed and vice versa. The role of these switches is to allow the process to make only one-conversation look-ahead.

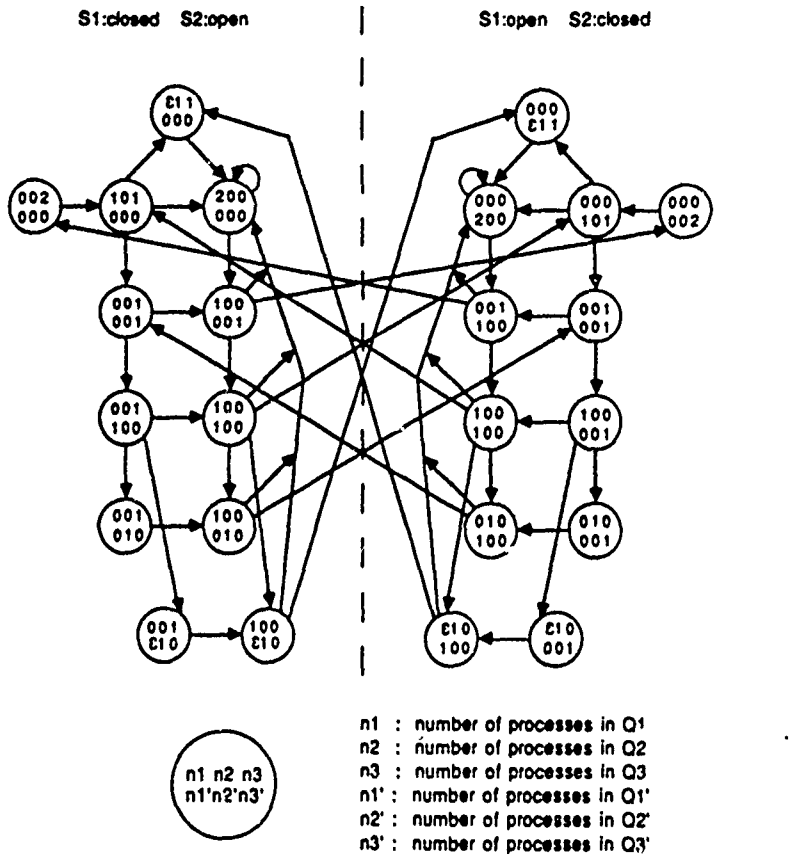


Fig. 8. State transition diagram for the two-process system operating under CL-1.

For example, assume that processes A and B in Fig. 3 start execution of $A1$ and $B1$, respectively, from $Q3$ in Fig. 7. Initially, switch $S1$ is closed (i.e., a process which has passed its acceptance test can go ahead through $S1$ instead of waiting in $Q2$) and switch $S2$ is open. At two different times the processes enter conversation CONV1 and thus move from $Q3$ to $Q1$. Now process B completes its conversation task $B2$ and passes its acceptance test while process A is still executing its conversation task $A2$. Under CL-0 process B should wait in $Q2$ until process A completes its conversation task. Under CL-1, however, process B skips $Q2$ (since $S1$ is closed) and proceeds into $Q3'$ to execute $B3$. If process B completes $B4$ in CONV2 and passes its acceptance test, then one of the following cases will arise.

Case 1: Process A is in $Q1$, i.e., still executes $A2$. In this case, process B should wait in $Q2'$ until process A completes $A2$ because only one-conversation look-ahead is allowed. The following two cases are possible later:

Case 1.1: Process A successfully completes CONV1. Process A enters $Q3'$. Now the status of the look-ahead switches becomes reversed, i.e., $S1$ is open and $S2$ is closed. Therefore, process B moves from $Q2'$ to $Q3$ and executes $B5$ immediately (Now processes in the upper queue set ($Q1$, $Q2$, $Q3$) are ahead of processes in the lower queue set ($Q1'$, $Q2'$, $Q3'$).)

Case 1.2: Process A fails the acceptance test of CONV1. Both processes roll back to $Q1$ and execute their alternate try blocks of CONV1. (All the look-ahead executions done by process B become nullified.)

Case 2: Process A has successfully completed CONV1 and already left $Q1$. The status of two switches was changed when process A left $Q1$ and entered $Q3'$. In this case, process B moves to $Q3$ without waiting in $Q2'$ because the look-ahead path is available.

Let ($n1$, $n2$, $n3$, $n1'$, $n2'$, $n3'$) represent a state of the queueing network in which $n1$, $n2$, $n3$, $n1'$, $n2'$, $n3'$ processes are in $Q1$, $Q2$, $Q3$, $Q1'$, $Q2'$, and $Q3'$, respectively, where $N = n1 + n2 + n3 + n1' + n2' + n3'$ and N is the number of customers (i.e., processes) moving through the network. Fig. 8 depicts the transition of the network among states when $N = 2$. The steady-state behavior of this network is discussed in Section IV.

3) Model for CL-2 (Two-Conversation Look-Ahead): A further extension of the model in Fig. 7 to represent the system operation under CL-2 results in the model depicted in Fig. 9. This queueing model contains three sets of three queues, i.e., queue sets ($Q1$, $Q2$, $Q3$), corresponding to the three queues in Fig. 5. There are three look-ahead switches, $S1$, $S2$, and $S3$, and only one switch is open at a time. Therefore, processes are allowed to make two-conversation look-ahead.

The state of this queueing network can be characterized by nine parameters, each representing the number of processes in a queue. The steady-state behavior of this network is discussed in Section IV.

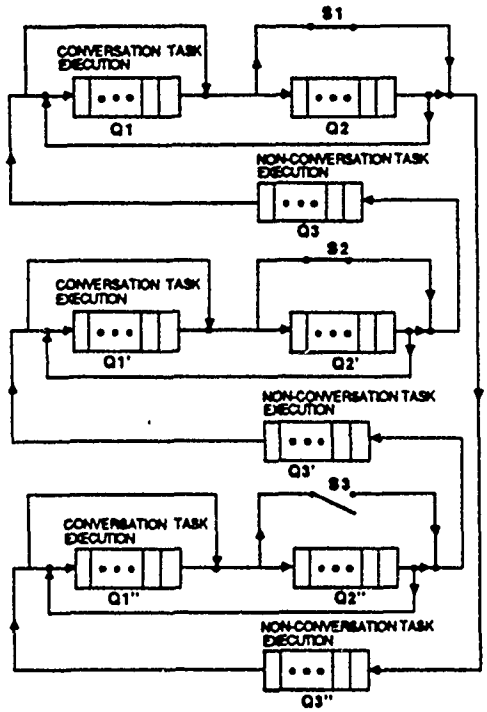


Fig. 9. A queueing network model for CL-2.

D. Analysis of the Queueing Network Models

The steady-state balance equations for all the states of the queueing network models formulated in the preceding section (III-C) are included in the Appendix. From the equations, the stationary probability of each state can be numerically obtained. In the next section, the performance of the system in parallel execution of conversations is analyzed by making use of such values.

The complexity of the analysis process grows rapidly as the look-ahead scope expands. A good indicator of this complexity is the number of states which a given system can be in. The exact number of all possible states of a system can be derived by use of the following formulas. Here n represents the total number of processes in the system.

1) CL-0

a) Nonerror states: $(n^2 + 3n)/2$
b) Error states: $n - 1$
c) Total: $(n^2 + 5n - 2)/2$

2) CL-1

a) Nonerror states: $(1/24) \times (n^4 + 10n^3 + 23n^2 + 14n)$
b) Error states: $(1/6) \times (n^3 + 3n^2 + 2n - 6)$
c) Total: $(1/24) \times (n^4 + 14n^3 + 35n^2 + 22n - 24)$.

3) CL-2

a) Nonerror states: $(1/720) \times (n^6 + 21n^5 + 145n^4 + 435n^3 + 574n^2 + 264n)$
b) Error states: $(1/120) \times (n^5 + 10n^4 + 35n^3 + 50n^2 + 24n - 120)$
c) Total: $(1/720) \times (n^6 + 27n^5 + 205n^4 + 645n^3 + 874n^2 + 408n - 720)$.

No. of Processes \ Model	CL-0	CL-1	CL-2
2 processes	6	12	18
6 processes	32	237	965
12 processes	101	2092	21111

Fig. 10. The numbers of states of queueing network models.

Fig. 10 summarizes the numbers of possible states for three different sizes of systems operating under three different execution schemes, CL-0, CL-1, and CL-2.

IV. Performance Comparison

In this section, various system performance indicators obtained through the analysis of the steady-state behavior of the models, developed in Section III-C, are discussed. Among the several performance indicators, the following are considered to be the most interesting ones:

1) *system throughput* evaluated in terms of the number of successful conversations per unit time,

2) *resource utilization* evaluated in terms of the number of processors idling due to the synchronization required inside the conversation, and

3) *conversation participation time*, i.e., the average amount of time a process spends inside each conversation.

A. System Throughput

The system throughput, TPc, indicated by the number of successful conversations per unit time, is obtained by use of the following formulas.

1) Under CL-0

$$TPc(0) = (1-\alpha) \times u1 \times P(1, N-1, 0)$$

where $(1 - \alpha)$ represents the probability that each process passes its acceptance test, and $u1$ represents the completion rate for the conversation task.

2) Under CL-1

$$TPc(1) = \sum_{n1'+n2'+n3'=N-1} (1-\alpha) \times u1 \times P(1, 0, 0, n1', n2', n3') + \sum_{n2'+n3'=N-1} (1-\alpha) \times u1 \times P(1, 0, 0, \epsilon, n2', n3')$$

3) Under CL-2

$$TPc(2) = \sum_{n1+n3+n1''+n2''+n3''=N-1} (1-\alpha) \times u1 \times P(n1, 0, n3, 1, 0, 0, n1'', n2'', n3'') + \sum_{n1+n3+n2''+n3''=N-1} (1-\alpha) \times u1 \times P(n1, 0, n3, 1, 0, 0, \epsilon, n2'', n3'') + \sum_{n2+n3=N-1} (1-\alpha) \times u1 \times P(\epsilon, n2, n3, 1, 0, 0, 0, 0, 0).$$

Figs. 11 and 12 depict the system throughputs under both

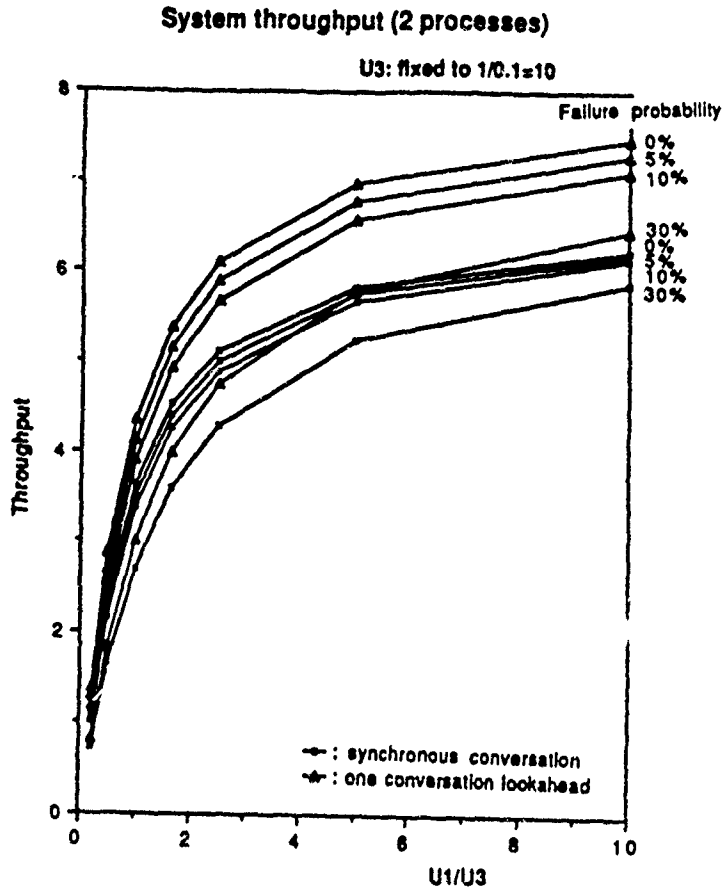


Fig. 11. System throughput under CL-0 and CL-1 (two processes).

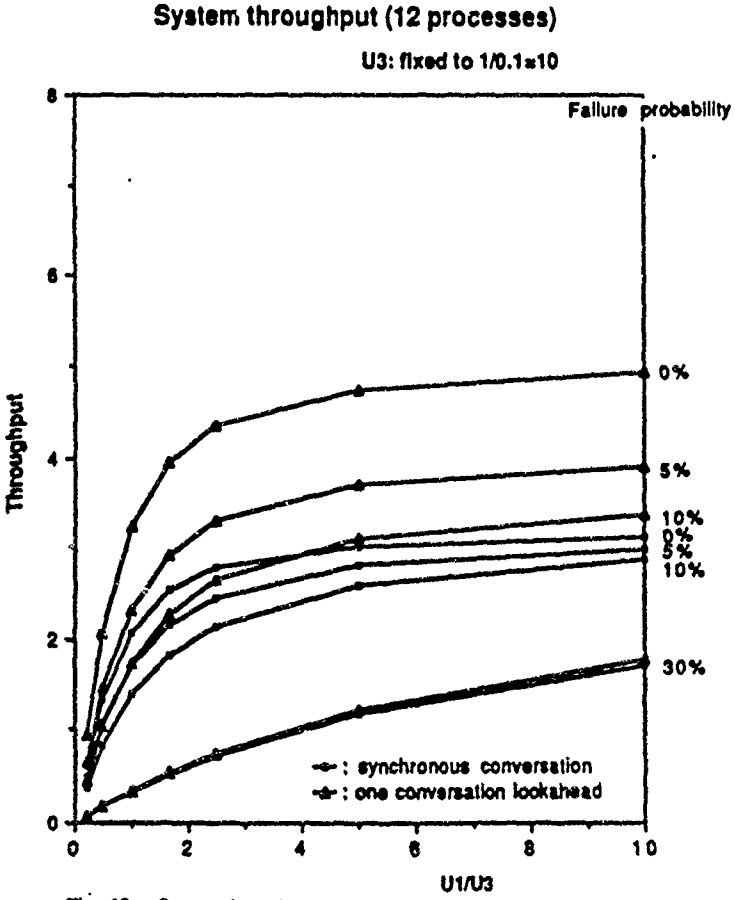


Fig. 12. System throughput under CL-0 and CL-1 (12 processes).

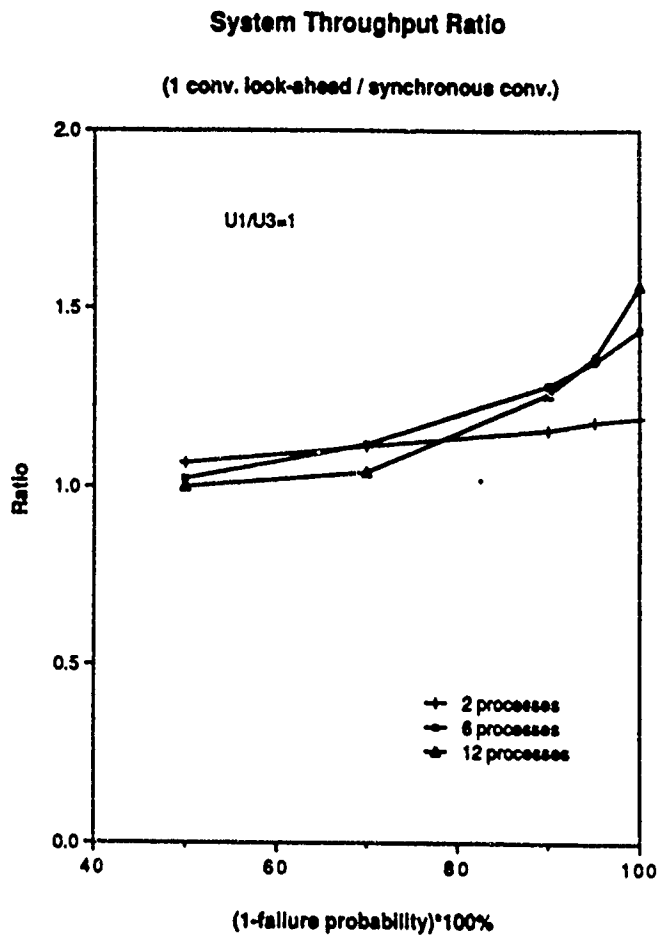


Fig. 13. System throughput ratio.

CL-0 and CL-1 for the cases of the number of processes being 2 and 12, respectively. The execution time ratio between the nonconversation task and the conversation task ($u1/u3$) varies from 0.1 to 10. The wide range of values for $u1/u3$ is examined because the ratio $u1/u3$ may actually vary widely among different applications. Since the execution time of the nonconversation task is fixed to 0.1 time unit, the execution time of the conversation task varies from 1 to 0.01 time unit. Each figure depicts for each conversation scheme (CL-0 or CL-1) four different curves corresponding to four different probabilities of acceptance test failure. Here we are interested only in those practical cases where the failure probability is less than 0.05.

Both figures show that under CL-0 the system throughput is not much affected by the probability of the acceptance test failure if there are a relatively small number of processes. However, the system throughput is more sensitive to the acceptance test failure probability under CL-1 than under CL-0. For example, when $u1/u3 = 10$ and the failure probability increases from 0 to 0.05, the system throughput for the case of 12 processes decreases by 0.1 under CL-0 whereas it decreases by 1.0 under CL-1 (Fig. 12). This is a reflection of the higher recovery cost under CL-1.

On the other hand, comparison of Fig. 11 and Fig. 12 reveals that as the number of conversation participants increase, the system throughput degrades more slowly under CL-1. For example, as the number of participants increases from 2 to 12 while $u1/u3 = 10$, the throughput degrades from 6.2 to 3.2 (i.e., 49 percent reduction) under CL-0 whereas the throughput degrades from 7.5 to 4.9 (i.e., 35 percent reduction) under CL-1. This also means that the benefits of look-ahead are greater when the number of participants is larger. Fig. 13 shows this phenomenon from another perspective. As long as the acceptance test failure probability is within a practical range, i.e., $0 < \alpha \ll 0.05$, the throughput increase resulting from incorporation of the single conversation look-ahead rule is greater when the number of participants is larger.

Fig. 14 shows the increase of system throughput resulting from the change of the scheme from CL-0 to CL-1 and then to CL-2. Two cases of the acceptance test failure probability, i.e., 0 and 0.05, are displayed and $u1/u3$ is 10. The figure shows that when the failure probability is as large as 0.05, the benefits of changing from CL-1 to CL-2 are not substantial, especially if the number of participants is six or more. On the other hand, when the failure probability is closer to zero, the benefits are substantial. For example, when the number of participants is six and the failure probability is zero, the benefit of changing from CL-0 to CL-1 is 46 percent increase in throughput whereas the benefit of changing from CL-0 to CL-2 is 69 percent increase in throughput. Therefore, CL-2 brings 23 percent additional increase in throughput in this case. The schemes permitting look ahead of three or more conversations can be analyzed in similar ways to support

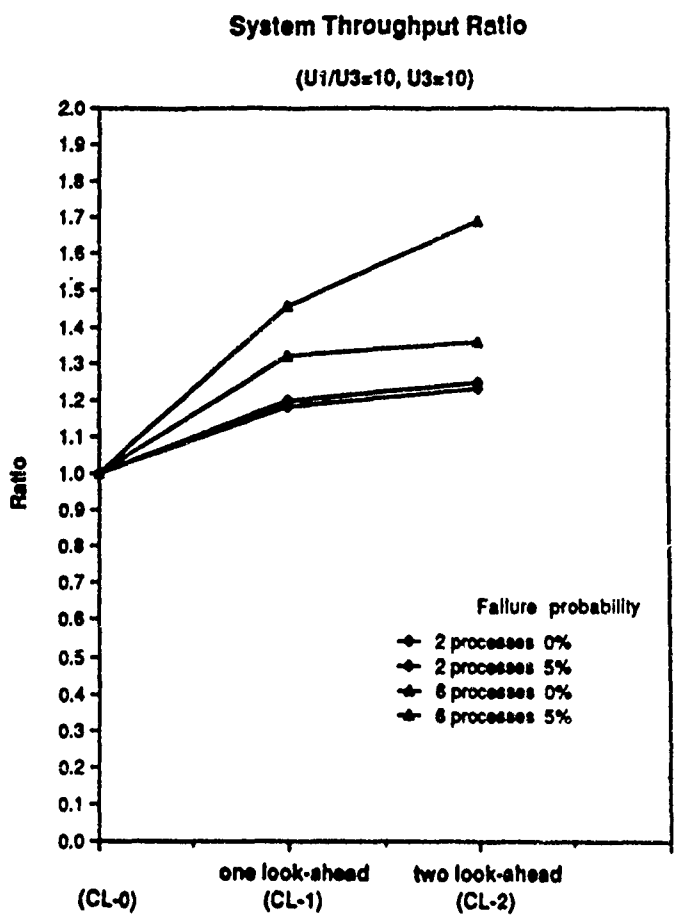


Fig. 14. System throughput ratio.

decisions regarding their adoption in given execution environments.

B. Resource Utilization

The number of processors idling due to the synchronization required on the processes inside the conversation is represented by the length of $Q2$ in the case of CL-0 and by the sum of the lengths of $Q2$ and $Q2'$ in the case of CL-1. The following formulas can be used to deerive the resource utilization.

1) Under CL-0

$$\mathrm{MQL2}(0) = \sum_{n2=1}^{N-1} \sum_{n1+n3=N-n2} n2 \times P(n1, n2, n3) + \sum_{n2=1}^{N-1} n2 \times P(\epsilon, n2, n3)$$

2) Under CL-1

$$\mathrm{MQL2}(1) = \sum_{n2'=1}^{N-1} \sum_{n1+n3+n1'+n3'=N-n2'} n2' \times P(n1, 0, n3, n1', n2', n3') + \sum_{n2'=1}^{N-1} \sum_{n1+n3+n3'=N-n2'} n2' \times P(n1, 0, n3, \epsilon, n2', n3') + \sum_{n2=1}^{N-1} n2 \times P(\epsilon, n2, n3, 0, 0, 0)$$

3) Under CL-2

$$\begin{aligned}\mathrm{MQL2}(2) = & \sum_{n2''=1}^{N-1} \sum_{n1+n3+n1'+n3'+n1''+n3''=N-n2''} n2'' \times P(n1, 0, n3, n1', 0, n3', n1'', n2'', n3'') \\ & + \sum_{n2''=1}^{N-1} \sum_{n1+n3+n1'+n3'+n3''=N-n2''} n2'' \times P(n1, 0, n3, n1', 0, n3', \epsilon, n2'', n3'') \\ & + \sum_{n2=1}^{N-1} \sum_{n3+n1'+n3'=N-n2} n2 \times P(\epsilon, n2, n3, n1', 0, n3', 0, 0, 0) \\ & + \sum_{n2'=1}^{N-1} n2' \times P(0, 0, 0, \epsilon, n2', n3', 0, 0, 0).\end{aligned}$$

Figs. 15 and 16 depict the queue lengths under both CL-0 and CL-1 for the cases where the number of processes are two and six, respectively. The probability of successful acceptance tests ($1 - \alpha$) varies from 0.5 to 1. The completion rate of the

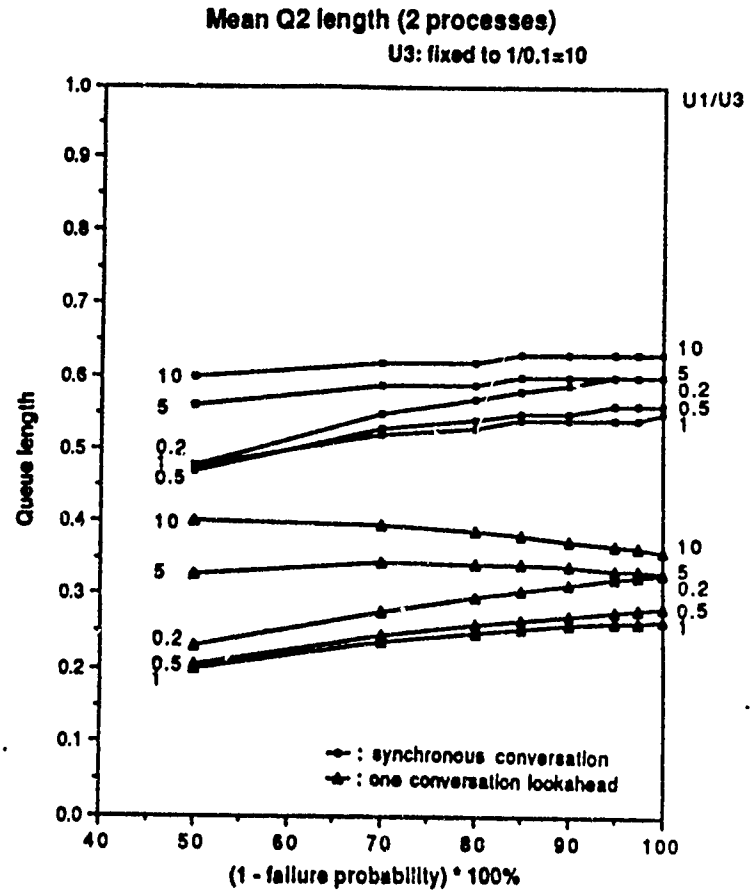


Fig. 15. Mean $Q2$ length under CL-0 and CL-1 (two processes)

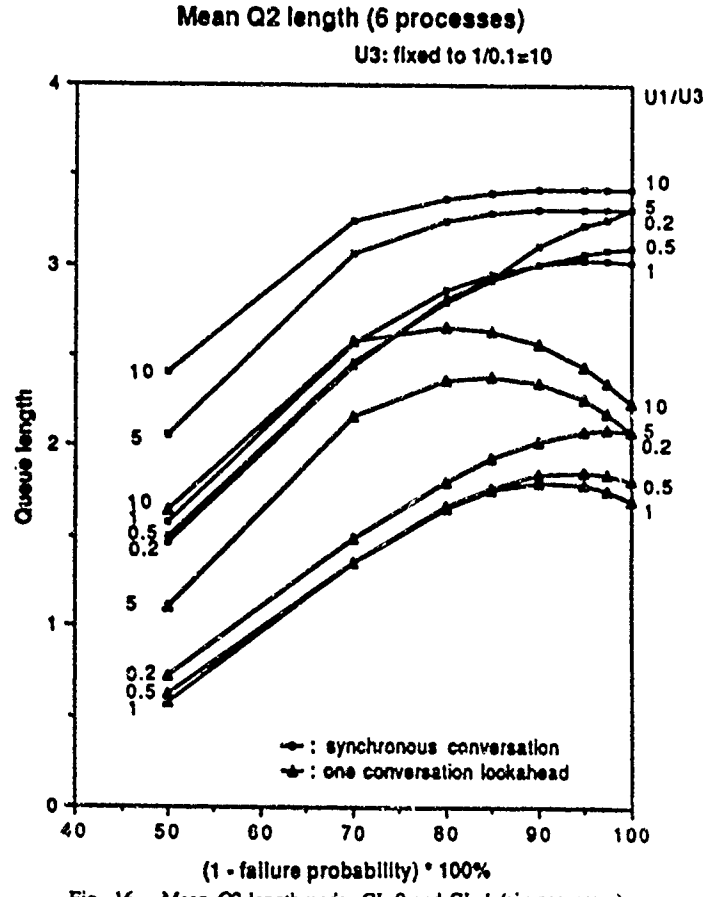


Fig. 16. Mean $Q2$ length under CL-0 and CL-1 (six processes).

nonconversation task ($u3$) was fixed at 10. Each figure depicts five different curves for each conversation scheme, each curve corresponding to a different execution time ratio ($u1/u3$).

As expected, processor utilization is substantially higher under CL-1 than under CL-0 when the failure probability is within a practical range ($0 < \alpha \ll 0.05$). When the failure probability is close to zero in a system of six processors and $u1/u3$ is 10, the expected percentage of busy processors is about 63 percent under CL-1 whereas it is only about 43 percent under CL-0. Also, under CL-0, processor utilization does not get much better as the failure probability approaches zero whereas it increases faster under CL-1.

Fig. 17 shows resource utilization in a six-processor system under CL-0, CL-1, and CL-2. Two different cases of $u1/u3$, 1 and 10, are shown. When $u1/u3$ is 10, the expected number of idling processors is 3.4 (57 percent of the processors are idle) under CL-0, 2.2 (37 percent) under CL-1, and 1.7 (28 percent) under CL-2.

C. Conversation Participation Time

Mean participation time, Wp, can be obtained by use of the Little's Law [11], i.e., $Wp = L/T$ where L is the queue length and T is the throughput of the queue server. Since the customers in $Q1$ (also $Q1'$, $Q1''$) and $Q2$ (also $Q2'$, $Q2''$) represent the processes inside a conversation, L is the sum of the lengths of those queues and T is $N \times$ TPc where N is the number of processes and TPc is the number of successful conversations per unit time discussed in Section IV-A.

1) Under CL-0

$$Wp(0) = (MQL1(0) + MQL2(0))/(N \times TPc(0))$$

2) Under CL-1

$$Wp(1) = (MQL1(1) + MQL2(1) + MQL1'(1) + MQL2'(1))/(N \times TPc(1))$$

3) Under CL-2

$$Wr(2) = (MQL1(2) + MQL2(2) + MQL1'(2) + MQL2'(2) + MQL1''(2) + MQL2''(2))/(N \times TPc(2)),$$

where MQL$m(n)$ denotes the length of Queue Qm in model CL-n and TPc(n) denotes the system throughput under CL-n.

Fig. 18 shows mean participation times under CL-0, CL-1, and CL-2 when the numbr of participating processes is six. Three different cases of failure probability are plotted. It shows that when the failure probability is within a practical range, the mean participation time improves significantly as the scheme changes from CL-0 to CL-1, but considerably less as the scheme changes from CL-1 to CL-2.

V. Summary

Overall the look-ahead approach reduces the synchronization overhead of the conversation scheme to a significant extent The system performance improves substantially by all three measures used in this paper Interestingly, as the synchronization overhead plays a less dominant role in determining the system performance under the look ahead approach (than under the basic conversation execution scheme), the impact of the acceptance test failure probability on the system performance is more noticeable. As the scope of look-ahead increases, the system performance improvement becomes gradually less substantial although implementation costs and recovery costs may grow steadily. Therefore, determination of a suitable limit on the scope of look-ahead requires a tradeoff analysis reflecting various environmental characteristics.

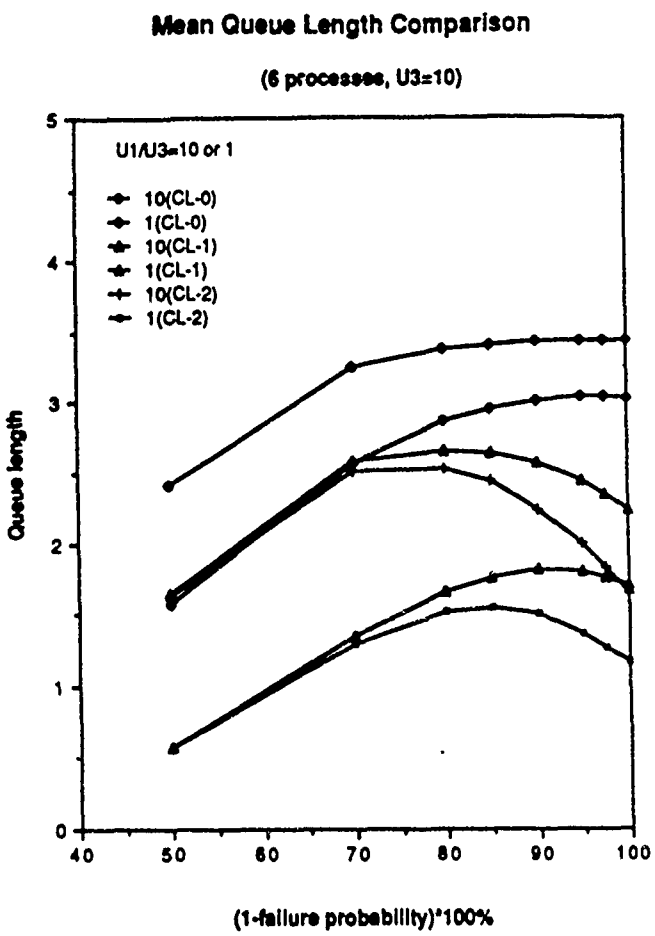


Fig. 17. Comparison of mean $Q2$ lengths under CL-0, CL-1, and CL-2.

The queueing network models developed in this paper can be extended to represent the systems in which processes engage in multiple types of conversations including some nested within others. For example, $Q3$ in Fig. 5 can be expanded into a series of $Q1$- and $Q2$-types representing different types of conversations followed by a queue of $Q3$-type in order to represent the systems in which processes engage in multiple types of nonnested conversations under CL-0. Such extended models will be useful in evaluating the potential performance of the systems being considered in specific application environments.

Formulation and analysis of the look-ahead approach to execution of conversations represent only one of many research tasks needed to establish the conversation scheme as a design technique that can be widely practiced. There are still several other fundamental questions regarding the conversation scheme, e.g., how to design effective conversation acceptance tests and alternate interacting sessions, that remain

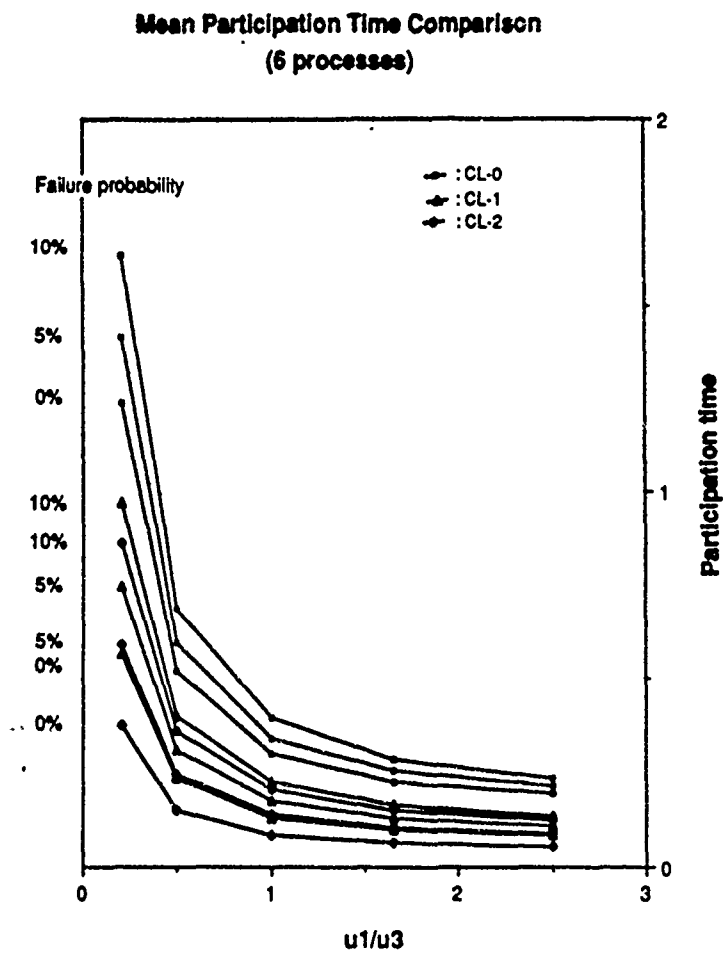


Fig. 18. Comparison of mean participation times under CL-0, CL-1, and CL-2.

unsatisfactorily answered. An experimental work aimed at finding answers to such questions and at validation of the predicted performances discussed here is regarded as a highly worthwhile subject for future research.

Appendix

Steady-State Balance Equations

The steady-state balance equations for the states of each queueing network model developed in Section III-C are as follows.

1) Model for CL-0

$$P(n1, n2, n3) = [1/(n1 \times u1 + n3 \times u3)] \times [u1 \times (n1+1) \times (1-\alpha) \times P(n1+1, n2-1, n3) + u3 \times (n3+1) \times P(n1-1, n2, n3+1)]$$

$$P(\epsilon, n2, n3) = [1/(n3 \times u3)] \times \left[\sum_{i=1}^{n2} u1 \times i \times \alpha \times P(i, n2-i, n3) + u3 \times (n3+1) \times P(\epsilon, n2-1, n3+1) \right]$$

2) Model for CL-1

$$P(n1, 0, n3, n1', n2', n3') = [1/((n1+n1') \times u1 + (n3+n3') \times u3)] \times [u1 \times (n1+1) \times (1-\alpha) \times P(n1+1, 0, n3, n1', n2', n3'-1) + u3 \times (n3+1) \times P(n1-1, 0, n3+1, n1', n2', n3') + u1 \times (n1'+1) \times (1-\alpha) \times P(n1, 0, n3, n1'+1, n2'-1, n3') + u3 \times (n3'+1) \times P(n1, 0, n3, n1'-1, n2', n3'+1)]$$

$$P(n1, 0, n3, \epsilon, n2', n3') = [1/(n1 \times u1 + (n3+n3') \times u3)] \times [u1 \times (n1+1) \times (1-\alpha) \times P(n1+1, 0, n3, \epsilon, n2', n3'-1) + u3 \times (n3+1) \times P(n1-1, 0, n3+1, \epsilon, n2', n3') + \sum_{i=1}^{n2'} u1 \times i \times \alpha \times P(n1, 0, n3, i, n2'-i, n3') + u3 \times (n3'+1) \times P(n1, 0, n3, \epsilon, n2'-1, n3'+1)]$$

$$P(\epsilon, n2, n3, 0, 0, 0)$$
$$= [1/(n3 \times u3)] \times \sum_{i=1}^{n2} \sum_{X=0}^{n2-i} u1 \times i \times \alpha$$
$$\times P(i, n2-i-X, n3, n1', n2', n3')$$
$$+ u3 \times (n3+1) \times P(\epsilon, n2-1, n3+1, 0, 0, 0)],$$
where $X = n1' + n2' + n3'$

(Note: The above steady-state balance equations cover only the case where switch $S1$ is closed and $S2$ is open in the model for CL-1. Due to the symmetric characteristics of this queueing network, it is not necessary to consider the other case where switch $S1$ is open and $S2$ is closed. Similarly, in the following, we consider only the case where switches $S1$ and $S2$ are closed and $S3$ is open in the model for CL-2.)

3) Model for CL-2

$$P(n1, 0, n3, n1', 0, n3', n1'', n2'', n3'')$$
$$= [1/((n1+n1'+n1'') \times u1 + (n3+n3'+n3'') \times u3)]$$
$$\times [u1 \times (n1+1) \times (1-\alpha)$$
$$\times P(n1+1, 0, n3, n1', 0, n3', n1'', n2'', n3''-1)$$
$$+ u3 \times (n3+1)$$
$$\times P(n1-1, 0, n3+1, n1', 0, n3, n1'', n2'', n3'')$$
$$+ u1 \times (n1'+1) \times (1-\alpha)$$
$$\times P(n1, 0, n3-1, n1'+1, 0, n3', n1'', n2'', n3'')$$
$$+ u3 \times (n3'+1)$$
$$\times P(n1, 0, n3, n1'-1, 0, n3'+1, n1'', n2'', n3'')$$
$$+ u1 \times (n1''+1) \times (1-\alpha)$$
$$\times P(n1, 0, n3, n1', 0, n3', n1''+1, n2''-1, n3'')$$
$$+ u3 \times (n3''+1)$$
$$\times P(n1, 0, n3, n1', 0, n3', n1''-1, n2'', n3''+1)]$$

$$P(n1, 0, n3, n1', 0, n3', \epsilon, n2'', n3'')$$
$$= [1/((n1+n1') \times u1 + (n3+n3'+n3'') \times u3)]$$
$$\times [u1 \times (n1+1) \times (1-\alpha)$$
$$\times P(n1+1, 0, n3, n1', 0, n3', \epsilon, n2'', n3''-1)$$
$$+ u3 \times (n3+1)$$
$$\times P(n1-1, 0, n3+1, n1', 0, n3', \epsilon, n2'', n3'')$$
$$+ u1 \times (n1'+1) \times (1-\alpha)$$
$$\times P(n1, 0, n3-1, n1'+1, 0, n3', \epsilon, n2'', n3'')$$
$$+ u3 \times (n3'+1)$$
$$\times P(n1, 0, n3, n1'-1, 0, n3'+1, \epsilon, n2'', n3'')$$
$$+ \sum_{i=1}^{n2'} u1 \times i \times \alpha$$
$$\times P(n1, 0, n3, n1', 0, n3', i, n2''-i, n3'')$$
$$+ u3 \times (n3''+1)$$
$$\times P(n1, 0, n3, n1', 0, n3', \epsilon, n2''-1, n3''+1)]$$

$$P(\epsilon, n2, n3, n1', 0, n3', 0, 0, 0)$$
$$= [1/(n1 \times u1 + (n3+n3') \times u3)]$$
$$\times \left[\sum_{i=1}^{n2} \sum_{Y=0}^{n2-i} u1 \times i \times \alpha \right.$$
$$\times P(i, n2-i-Y, n3, n1', 0, n3', n1'', n2'', n3'')$$
$$+ u3 \times (n3+1)$$
$$\times P(\epsilon, n2-1, n3+1, n1', 0, n3', 0, 0, 0)$$
$$+ u1 \times (n1'+1) \times (1-\alpha)$$
$$\times P(\epsilon, n2, n3-1, n1'+1, 0, n3', 0, 0, 0)$$
$$+ u3 \times (n3'+1)$$
$$\left. \times P(\epsilon, n2, n3, n1'-1, 0, n3'+1, 0, 0, 0) \right]$$
where $Y = n1'' + n2'' + n3''$

$$P(0, 0, 0, \epsilon, n2', n3', 0, 0, 0)$$
$$= [1/(n3' \times u3)]$$
$$\times \left[\sum_{i=1}^{n2'} \sum_{Z=0}^{n2'-i} u1 \times i \times \alpha \right.$$
$$\times P(n1, 0, n3, i, n2'-i-Z, n3', n1'', n2'', n3'')$$
$$+ u3 \times (n3'+1)$$
$$\left. \times P(0, 0, 0, \epsilon, n2'-1, n3'+1, 0, 0, 0) \right],$$

REFERENCES

[1] B. Bhargava, Ed., *Concurrency and Reliability in Distributed Systems*. New York: Van Nostrand and Reinhold, 1987.

[2] R. H. Campbell, T. Anderson, and B. Randell, "Practical fault tolerant software for asynchronous systems," in *Proc. IFAC Safecomp 83*, 1983, pp. 59–65.

[3] C. G. Davis and R. L. Couch, "Ballistic missile defense: A supercomputer challenge," *IEEE Computer*, pp. 37–46, Nov. 1980.

[4] E. C. Foudriat, *et al.*, "An operating system for future aerospace vehicle computer systems," NASA Tech. Memo. 85784, Apr. 1984.

[5] S. T. Gregory and J. C. Knight, "A new linguistic approach to backward error recovery," in *Proc. FTCS-15*, 1985, pp. 404–409.

[6] H. Hecht, "Fault tolerant software for real-time applications," *Comput. Surveys*, pp. 391–407, Dec. 1976.

[7] K. H. Kim, D. L. Russell, and M. J. Jenson, "Language tools for fault-tolerant programming," Tech. Memo. PETP-1, Electon. Sci. Lab., USC, Nov. 1976.

[8] ——, "Approaches to mechanization of the conversation scheme based on monitor," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 189–197, May 1982.

[9] K. H. Kim, S. M. Yang, and M. H. Kim, "Implementation of concurrent programming language facilities supporting conversation structuring," in *Proc. 1985 COMPSAC, Int. Comput. Software Appl. Conf.*, Oct. 1985, pp. 445–453.

[10] K. H. Kim, S. Heu, and S. M. Yang, "An analysis of the execution overhead inherent in the conversation scheme," in *Proc. 5th Symp. Reliability Distribut. Software Database Syst.*, Jan. 1986, pp. 159–168.

[11] L. Kleinrock, *Queueing Systems Vol. I: Theory*. New York: Wiley, 1975.

[12] W. C. McDonald and R. W. Smith, "A flexible distributed testbed for real-time applications," *IEEE Computer*, pp. 25–39, Oct. 1982.

[13] B. M. Ozake, E. B. Fernandez, and E. Gudes, "Software fault

tolerance in architectures with hierarchical protection levels," *IEEE Micro*, vol. 8, pp. 30–43, Aug. 1988.
[14] C. V. Ramamoorthy *et al.*, "Application of a methodology for the development and validation of reliable process control software," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 537–555, Nov. 1981.
[15] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220–232, June 1975.
[16] D. L. Russell, and M. J. Tiederman, "Multiprocess recovery using conversations," in *Proc. FTC-9*, 1979, pp. 106–109.
[17] S. K. Shrivastava, L. Mancini, and B. Randell, "On the duality of fault tolerant system structures," Tech. Memo. SRM/455, Computing Lab., Univ. of Newcastle upon Tyne, 1987.
[18] W. N. Toy, "Fault-tolerant computing," in *Advances in Computers*. vol. 26. New York: Academic, 1987, pp. 201–279.
[19] S. M. Yang, "Timing specification and verification for fault-tolerant distributed computer systems," Ph.D. dissertation, Dep. Comput. Sci. Eng., Univ. of South Florida, Dec. 1986.

K. H. Kim (S'73–M'75–SM'86–F'89) received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1969, the M.A. degree in computer science from the University of Texas, Austin, in 1972, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1974.

From 1969 to 1971 he served as an officer the the Korean Army. From 1975 to 1986 he served on the faculty of the University of South Florida, Tampa, the State University of New York, Binghamton, and the University of Southern California, Los Angeles. While teaching at Binghamton, he also served as acting chairman of the Department of Computer Science for nine months. He is currently a Professor of Computer Engineering in the Department of Electrical Engineering at the University of California, Irvine. His current research interests are in the areas of reliable distributed processing and real-time software engineering. He is currently conducting both analytic and experimental research in the DREAM (Distributed Real-Time Ever Available Microcomputing) Laboratory.

Dr. Kim is a fellow of the IEEE Computer Society and a member of the Association for Computing Machinery and the IFIP Working Group 10.4. He served as the Chairman of the IEEE Computer Society's Technical Committee on Distributed Processing from September 1984 to December 1986. In 1989 he plans to host the IEEE Computer Society's 9th International Conference on Distributed Computing Systems as the General Chairman.

Seung M. Yang (S'82–M'86) received the B.S. degree in electronics from the Seoul National University, Seoul, Korea, in 1978, and the M.S. and Ph.D. degrees in computer science from the University of South Florida, Tampa, in 1983 and 1986, respectively.

He is currently an Assistant Professor in the Department of Computer Science Engineering at the University of Texas at Arlington, Arlington. He worked for Samsung Semiconductor and Telecommunication Corporation, where he was involved in development of electronic switching systems during the period of 1978 and 1981. His research interests are in the areas of reliable distributed computing, real-time software engineering, and parallel systems.

Appendix A.VI

Implementation of the Conversation Scheme in Loosely Coupled Distributed Computer Systems

Implementation of the Conversation Scheme in Loosely Coupled Distributed Computer Systems

S. M. Yang
Computer Science Engineering Department
University of Texas at Arlington
Arlington, TX 76019

K. H. Kim
Computer Engineering Program
Department of Electrical Engineering
University of California
Irvine, CA 92717

ABSTRACT

This paper discusses several different approaches for implementing conversations in loosely coupled distributed computer systems (DCS's). Important implementation factors to be considered include the control of exits of processes upon completion of their conversation tasks and the approach to execution of the conversation acceptance test. Two different exit control strategies, one in a synchronous manner and the other in an asynchronous manner, and three different approaches to execution of the conversation acceptance test, centralized, decentralized, and semi-centralized, are examined and compared in terms of system performance and implementation cost. The effectiveness of these execution approaches also depends on the way conversations are structured initially by program designers. Therefore, the two major types of conversation structures, Name-Linked Recovery Block (NLRB) and Abstract Data Type (ADT) Conversations, are examined to analyze which execution approaches are the most efficient for each conversation structure. These results provide useful guidelines for implementing conversations in loosly coupled DCS's.

Index Terms: fault-tolerant cooperating processes, conversation, acceptance test, recovery line, loosely coupled network, lookahead.

1. Introduction

Since the concept of *conversation structuring* was introduced in an abstract form as an approach to facilitating cooperative recovery in systems of interacting processes [Ran75], continuous research efforts have been invested to convert the concept into a practical technology. For example, several mechanized structuring schemes containing practical language syntax and associated precise semantics have been formulated [Cam83,Gre85,Kim82,Rus79]. Subsequently, some practical language processing systems have been produced [Kim85]. The execution costs of conversations have also been studied [Kim86,Kim88].

The work reported here was supported in part by the Office of Naval Research under Contract No. N00014 87-K-0231 and Contract No. N00014-88-K-0622, in part by U.S. Army SDC and NASA JPL under Contract No. NAS7-918-RE182/443, and in part by University of California and AT&T under MICRO Grant No. 88-123.

However, experimental study of the scheme has been scarce and has not yet progressed to the point of producing concrete results. Moreover, techniques for efficient implementation, especially in distributed computer systems (DCS's), have not been much studied either. This paper presents issues in implementing conversations in DCS's together with some efficient implementation approaches. The following three cost factors should be carefully considered in such implementation.

The first factor is the cost of communication between remote processes. In selecting an efficient implementation strategy of the conversation scheme, this interprocess communication cost plays an important role as will be elaborated in this paper. The communication cost is primarily determined by the network structure adopted, i.e., network topology, communication medium, and protocol used. In this paper DCS's in which computing nodes are geographically dispersed beyond a single room, i.e., loosely coupled network (LCN) systems, are considered.

The second factor is the cost of synchronizing processes at the exit of a conversation. As shown in [Kim86], this cost can have significant impact on the performance of the conversation scheme. To reduce this cost, the approach of conversation lookahead, i.e., to allow asynchronous exits from the conversation, was studied in [Kim88]. The conversation scheme extended with the lookahead capability is called the *asynchronous conversation* scheme whereas the conversation scheme with no lookahead is called the *synchronous conversation* scheme. Under the synchronous conversation scheme, all processes are synchronized at the end of each conversation, i.e., no process is allowed to exit from the conversation earlier than other processes even if the process has passed its acceptance test. On the other hand, under the asynchronous conversation scheme processes are allowed to exit from the conversation as they finish their tasks.

The third factor is the cost of designing and executing the *conversation acceptance test* (CAT). In a single-node/multiprocess system, the choice between the centralized CAT (i.e., one participant takes care of the global acceptance test routine) and the decentralized CAT (i.e., each participant has its own acceptance test routine), has relatively insignificant effect on the system performance. However, in an LCN system, the CAT choice affects the system performance to a significant extent. In this paper, three different approaches to design and execution of CAT's, *centralized*, *decentralized*, and *semi-centralized*, are discussed.

In the next section, approaches to implementing synchronous and asynchronous exit control strategies are introduced and pros and cons of each approach are discussed. In Section 3, three different CAT execution approaches are presented. The effectiveness of different execution approaches also depends on the way conversations are structured by program designers. The issue of using two different basic types of conversation structures, *Name-Linked Recovery Block (NLRB)* and *Abstract Data Type (ADT) Conversation*, are examined in Sections 4 and 5 respectively, in order to analyze the effectiveness of the execution approaches discussed earlier. Section 6 is the summary section.

2. Synchronous and Asynchronous Conversations

This section starts with a brief description of the conversation structure. Then the characteristics of synchronous and asynchronous conversations are discussed. These two approaches are quite different in terms of their impacts on system performance and complexity. A brief comparison is made at the end of this section.

2.1 Basic logical structure of the conversation

The conversation is a two-dimensional enclosure of recoverable activities of multiple interacting processes, in short, a recoverable interacting session [Kim82, Ran75]. It creates a "boundary" which process interactions may not cross. The boundary of a conversation consists of a recovery line, a test line and the walls defining membership as shown in Figure 1. Each participant process contains one or more try blocks designed to produce the same or similar computational results as well as an acceptance test which is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A recovery line is a coordinated set of the recovery points of interacting processes that are established (possibly at different times) before interactions begin. A test line is a correlated set of the acceptance tests of interacting processes. A conversation is successful only if all the interacting processes pass their acceptance tests forming the test line. Therefore, the participants are allowed to leave the conversation when all the participants have passed their acceptance tests. If any of the acceptance test fails, all the processes roll back to the recovery line and retry with their alternate try blocks. These alternate try blocks collectively define an alternate interacting session (AIS) where as the set of primary try blocks executed first after the processes enter the conversation define the primary interacting session (PIS). A process that has executed its try block and passed its acceptance test is said to have finished its conversation task. A process which is inside a conversation cannot interact with a process which is not in the conversation. Conversations must be strictly nested in two dimensions. That is, when conversation C.nest is nested within conversation C, the set of processes that participate in nested conversation C.nest must be a subset of the processes that participate in C, the entire recovery line of C.nest must be established after the entire recovery line of C, and the entire test line of C.nest must be set before the entire test line of C.

2.2 Exit control

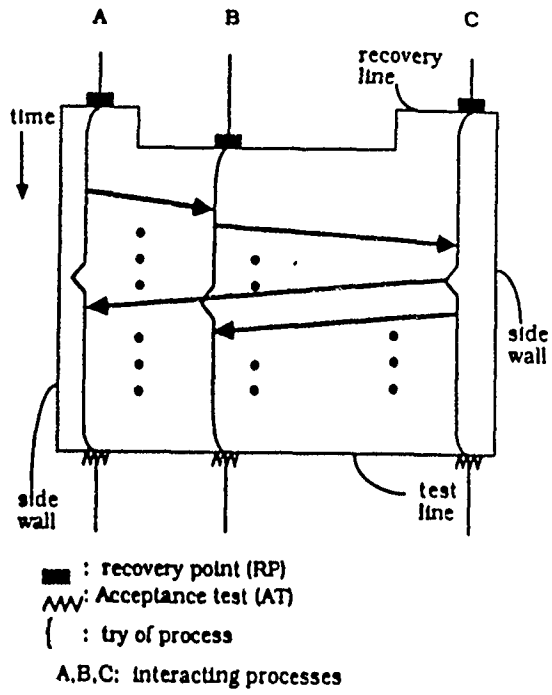
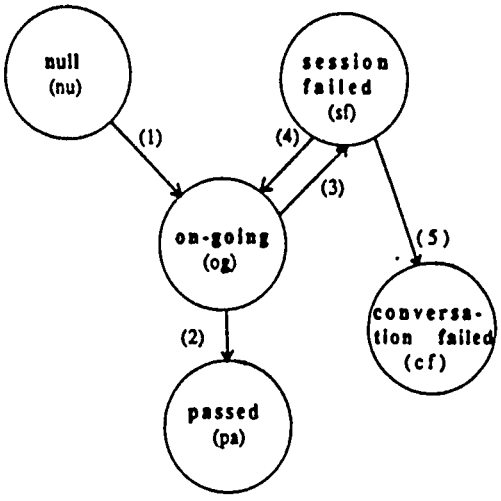


Figure 1. Conversation.

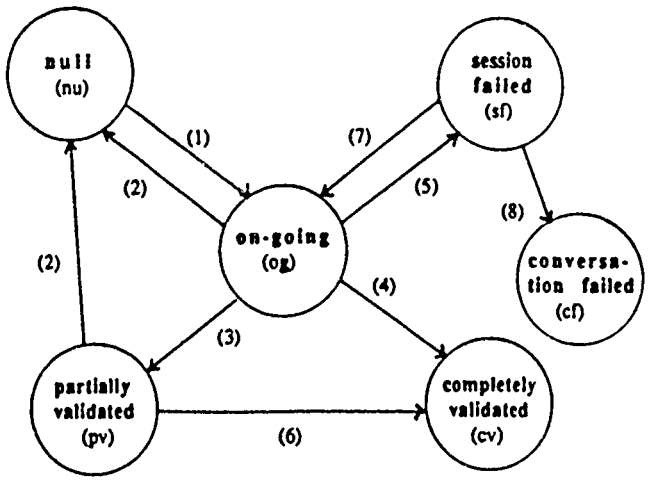
In the basic conversation scheme sketched above, processes enter a conversation asynchronously but synchronize themselves before exiting from it. The synchronization considered here is of a special kind specifically required by the conversation scheme and thus different from the application-dependent synchronization required between cooperating processes. The synchronization can add significantly to the time cost of the conversation scheme. A fundamental approach to reducing the synchronization overhead is the lookahead. Under the lookahead approach each participant process leaves the conversation as soon as it passes its own acceptance test. It does so with the awareness of the possibility that another participant may execute an acceptance test later and fail in the acceptance test thus making it necessary for the former to roll back to the recovery line of the conversation. Therefore, the lookahead approach here is an optimistic approach and is aimed at trading increase in recovery costs for reduction of synchronization overhead.

Although it is logically feasible to make provisions for a participant process to execute beyond the unfinished conversation to an unlimited extent, it is practical to limit the extent of the lookahead with respect to controlling the implementation complexity. In addition the lookahead should not be allowed to go past the points where critical irreversible actions, e.g., certain critical output actions, are taken. In this paper, the cases where lookahead is allowed to limited extents are considered.



(1) Start execution of the primary interacting session
(2) All the participants have passed their acceptance tests
(3) A participant has failed in its acceptance test
(4) Start execution of the next alternate interacting session
(5) A conversation has failed due to the failures of all interacting sessions

Figure 2. Change in the state of a synchronous conversation.



(1) Start execution of the primary interacting session
(2) Conversation execution is nullified due to the rollback of a participant to an older conversation
(3) All the participants have passed their acceptance tests but there is at least one participant which is in "revocable" state
(4) All the participants have passed their acceptance tests and are "irrevocable"
(5) A participant has failed in its acceptance test
(6) All the participants have become "irrevocable"
(7) Start execution of the next alternate interacting session
(8) A conversation has failed due to the failures of all interacting sessions

Figure 3. Change in the state of an asynchronous conversation.

The conversations which are executed under the lookahead-permitting strategy are called the *asynchronous conversations* whereas those executed where the lookahead is not permitted, i.e., only the synchronous exit of participant processes is facilitated, are called the *synchronous conversations*. Although the concepts of synchronous and asynchronous conversations were introduced in earlier papers [Kim76, Kim86,Rus79], their implementation techniques were not studied in depth. Finite state machine representations of both conversations that can be useful in implementing the conversations are provided below.

A synchronous conversation may transit among the following five different states:

(S1) null: None of the participants have entered into the conversation.
(S2) ongoing: There is at least one participant which is executing its conversation task.
(S3) passed: All the participants have passed their acceptance tests.
(S4) session failed: At least one participant has failed in its acceptance test.
(S5) conversation failed: The conversation has failed due to the failures of all alternate interacting sessions.

Figure 2 shows possible changes in the state of a synchronous conversation. Once a participant process fails in its acceptance test (transition (3) in the figure), all the participant processes abandon the try with the current interacting session and later start a retry with the next alternate interacting session.

In the case of an asynchronous conversation, a process that has exited from a conversation C1 via lookahead may enter another conversation C2. If some slow progressing participants of C1 are not participants of C2, then it is possible that C2 activities including acceptance tests are completed while C1 remains unfinished. In such a case, C2 should be treated as an unfinished conversation until C1 becomes completed. This is because if C1 fails, then all C2 activities that have taken place must be nullified as a part of the rollback to the recovery line of C1.

Definition 1: A conversation is validated if all the participants of the conversation have passed their acceptance tests.

Definition 2: A process is irrevocable if all the conversations which the process has participated in have been validated.

Definition 3: A conversation is completely validated if it is validated and all participant processes are irrevocable. On the other hand, a conversation is partially validated if it is validated but at least one participant remains revocable.

An asynchronous conversation may, therefore, transit among the following six different states as shown in Figure 3.

(S1) null: None of the participants have entered into the conversation.

(S2) on-going: There is at least one participant which is executing the conversation task.
(S3) partially validated:
(S4) completely validated:
(S5) session failed: At least one participant has failed in its acceptance test.
(S6) conversation failed: The conversation has failed due to the failures of all interacting sessions.

In Figure 3 the transition (2) means that the on-going or partially validated conversation (say X) is nullified when one of the participants must roll back to an older conversation (say Y). If a retry of the older conversation (Y) is successful, the participants may again execute the conversation (X) with their primary try blocks (transition (1) in the figure).

2.4 Discussion

Intuitively, the system performance would increase if asynchronous exits of the participants are allowed. The performance improvement would be particularly conspicuous when the acceptance test failure probability is very low (making the rollback infrequent) and the number of participants is large (making the synchronization overhead substantial).

An analytic study on the performance of the conversation scheme based on a queueing network model [Kim86,Kim88] showed that the performance of a system would be significantly affected by the synchronization required of the processes in exiting from a conversation, not by the failure probability of each process. For example, suppose a process is allowed to make "one conversation lookahead", which means that a process can continue lookahead as long as it has not exited from more than one unfinished (i.e., on-going or partially validated) conversation. The analytic study showed that by allowing one conversation lookahead the performance could increase about 46% when the number of participants is six, the failure probability is almost zero, and the amount of computation that a process performs inside a conversation is on the average about 10% of the total computation the process performs. On the other hand, the performance would decrease only 2% when the failure probability increases from zero to 0.05 (which is higher than that of most practical systems) and other parameters including synchronous exit remain unchanged.

Implementation of an asynchronous conversation, however, would be costly. Each process keeps its own history of the conversations which have not been completely validated. The history of each conversation includes the rollback point, values of the global variables which have been changed after the process entered the conversation, names of other participants, etc. Therefore, a new design parameter introduced that should be chosen through a tradeoff analysis is the number of unfinished conversations a process is allowed to be associated with.

3. Conversation Acceptance Test

This section describes three different approaches to execution of the conversation acceptance test (CAT): centralized CAT, decentralized CAT, and semi-centralized CAT. Advantages and disadvantages of each approach are also discussed.

3.1 Centralized CAT

In this approach only one designated participant, named "head" participant, contains the complete CAT routine. Therefore, the head participant executes the CAT when all the participants finish their execution of try blocks and then broadcast the CAT result to other participants. Since the code that implements the CAT is not scattered among the participant processes this approach has an advantage of not requiring the decomposition of the CAT routine. This property is valuable where the CAT is designed as a single function.

However, the centralized CAT approach may lead to relatively large communication overhead due to the sizable messages sent from the participants to the head participant. The messages include the values of the variables needed for CAT. Another deficiency of this approach is that the malfunctioning of the head participant process causes the loss of the entire CAT function. Various ways to avoid such a loss of the CAT function are conceivable but they all represent additional costs.

3.2 Decentralized CAT

In this approach each participant performs its own acceptance test, and the participants exchange their results with each other. Therefore, every process receives the results of other participants and figures out the result of the "global" acceptance test.

One of the advantages of this approach is that all participants have the symmetric structure. This makes it simple to implement. Also, treatment of the malfunctioning participant may be easier than in the centralized CAT case. However, decomposition of the CAT function is sometimes a costly burden on the programmer. Although automated decomposition is conceivable, its practicality requires further study. Also, if an efficient broadcasting channel is not available, the number of CAT-related messages exchanged among participants may become very large, although each message will be short. Therefore, in such an environment the communication cost becomes very high.

3.3 Semi-centralized CAT

This approach compromises the above two approaches in such a way that the "local" acceptance test is done by each participant and the CAT result is determined by the head participant. That is, each participant performs its own acceptance test and sends the result to the head participant. Then the head participant judges the success or failure of the CAT depending upon whether all the reports received are success reports or not, and then broadcasts the CAT result.

The main advantage of this approach is the small communication overhead. This will be further elaborated in the next section (3.4). The semi-centralized approach, however, shares one deficiency with the decentralized CAT approach. That is, it is necessary to decompose the CAT function.

3.4 Discussion

Table 1 summarizes the number of messages needed for CAT under each approach. As shown in the table

the total number of messages is the same for all three cases, but the messages communicated under the three approaches are different in length and number of destination processes. For example, the messages from the participants to the head participant under the centralized CAT approach would be sizable because the messages include the values of the variables needed for CAT. On the other hand, the CAT-related messages required in the semi-centralized and decentralized CAT approaches include "pass" or "fail" information only. Therefore, the size of the message is very short and fixed. Nevertheless, all the messages in the decentralized CAT approach need to be broadcast whereas only one message (the global CAT result message broadcast by the head participant) needs to be in the centralized and semi-centralized CAT approaches. It seems reasonable to conclude that the semi-centralized CAT approach is the best in terms of message traffic.

However, each approach has its own advantages and deficiencies. The choice is mainly related to the following two factors: (1) characterisitics of the application program such as process structure, reliability consideration, etc., and (2) characteristics of the communication system such as network topology, communication medium, and protocol used. Also, the conversation structuring scheme used by the program designer favors a certain CAT execution approach. This will be further elaborated in the remainder of the paper.

4. Implementation of the NLRB Scheme in LCN Systems

This section and the following section describe how some mechanized structuring schemes used by the program designer and the execution approaches discussed in preceding sections can be used in combinations in implementing conversations in LCN systems. The structuring schemes selected for discussion in this paper are the *Name-Linked Recovery Block (NLRB)* scheme and the *Abstract Data Type (ADT-) Conversation* scheme described in [Kim82,Rus79].

LCN systems considered in these two sections have the following characteristics:
(1) Each node contains a processor, a local memory, and an IO handler.
(2) Each node runs one or more software processes.
(3) Communication between processes in different nodes is done through a bus with the aid of IO handlers. (Broadcasting is facilitated).
(4) The IO handler in the destination node receives and puts the message into the mailbox of the destination process.

4.1 NLRB scheme

The basic idea of the NLRB scheme is to extend the recovery block (RB) construct with a conversation identifier field as follows [Kim82,Rus79]:

```
[ conv C:]
ensure      T
by          B1
else by     B2
...
else by     Bn
else error
```

Message type \ Policy	Centralized	Decentralized	Semi-centralized
Total number of messages for CAT	$N(l)+1(sn)$	$(N+1)(sn)$	$N(s)+1(sn)$
1-to-1 large* messages (l)	N		
1-to-1 small messages (s)			N
1-to-n small messages (sn)	1	N+1	1

Number of participants = N+1

* This message contains the values of variables needed for CAT

Table 1. Comparison in number of messages for CAT.

where C is the conversation identifier, T the acceptance test, and Bi, 1<i<n, the try blocks. The set of name-linked RB's, each executed by a different process but having the same conversation identifier, compose a conversation construct.

Although this scheme has several deficiencies as pointed out in [Kim82], the scheme is believed to be suitable in some LCN environments for the following reason. In LCN systems, nodes are geographically dispersed. It is very likely that in some application environments processes on different nodes are designed independently. In such an environment the NLRB scheme is more natural for use than other schemes such as the ADT-Conversation scheme [Kim82].

4.2 Syntax adopted and message format

The following syntax is an extension of the original NLRB syntax and the extended part is the "participant" field.

```
[ conv C:]
participants   PROCA, PROCB, ...
ensure         T
by             B1
else by        B2
...
else by        Bn
else error
```

where PROCA, PROCB, ... are the process ids of the conversation participants. (Each process has a unique process id in the system.)

In the following subsections, execution of NLRB's under the two different exit control approaches and the two different CAT execution approaches, semi-centralized and decentralized are considered. The centralized CAT approach was not considered because each participant of the NLRB conversation is designed to have its own acceptance test routine and placing all the participants' acceptance test routines in one node did not seem competitive with other two approaches in terms of resulting execution performance.

ML: message length (number of bytes)
T: message type
N: number of destination processes
D: destination process id's
S: source process id
Data: contents
EOM: end of message

Figure 4. Message format.

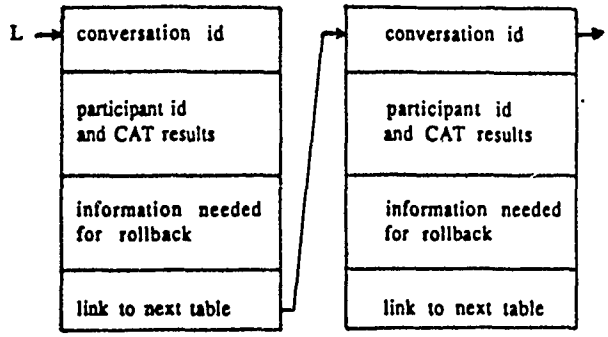


Figure 5. Conversation Table.

The generic message format adopted is shown in Figure 4. Note that it is possible to send the message to more than one destination process at a time. By doing this the number of messages communicated among the processes can be reduced. Messages are classified largely into two types: normal message and CAT result message. In the case of a CAT result message, the data field is empty. As will be shown in the following subsections, different types of CAT result messages are required for different implementations.

4.3 Approach N-1 for execution of NLRB's: Synchronous exit and semi-centralized CAT execution

Under the synchronous exit approach, history logging is not necessary because a participant can exit the conversation only when all the participants pass their acceptance tests. Therefore, its implementation is relatively simple.

The types of CAT result messages used in this execution approach are as follows:
(1) success notice (SU): This message is sent to the head participant when the participant has passed its local acceptance test.
(2) failure notice (FA): This message is sent to the head participant when the participant has failed in its local acceptance test.
(3) global success notice (GS): This message is broadcast when the head participant has concluded the CAT to be a success.
(4) global failure notice (GF): This message is broadcast when the head participant has concluded the CAT to be a failure.

4.4 Approach N-2 for execution of NLRB's: Synchronous exit and decentralized CAT execution

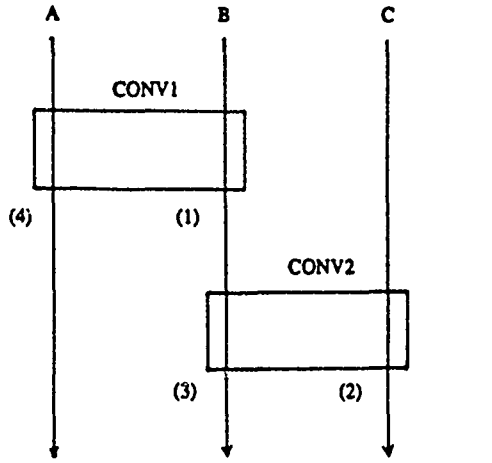
The execution approach is almost the same as the previous one (Approach N-1). As mentioned earlier, however, there is no head participant with the decentralized CAT approach. The success or failure notice from each participant is broadcast to all other participant processes. Besides these two no other types of CAT result messages are required.

4.5 Approach N-3 for execution of NLRB's: Asynchronous exit and semi-centralized CAT execution

Under the asynchronous exit approach, the information needed to roll back, e.g., rollback point, values of the global variables, etc., should be kept until the conversation is completely validated. In order to keep this information each process maintains a table called the "conversation table" for each unfinished conversation. The entries of the conversation table are shown in Figure 5. A conversation table is created when the process initially enters the conversation and removed when the conversation becomes completely validated. These tables are linked as they are created, i.e., the table of the newly executed conversation is attached to the end of the linked list.

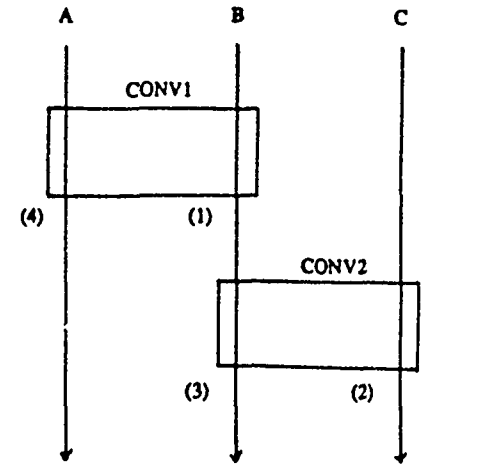
The types of CAT result messages needed are more complicated than in the case of the synchronous exit:
(1) complete local validation notice (CV): This message is sent to the head participant when a process that is in the irrevocable state has passed its acceptance test.
(2) partial local validation notice (PV): This message is sent to the head participant when a revocable process has passed its acceptance test.
(3) failure notice (FA): This message is sent to the head participant when a process has failed in its acceptance test.
(4) global success notice (GS): This message is broadcast when the head participant has concluded the CAT to be a success. That is, the conversation is completely validated.
(5) global failure notice (GF): This message is broadcast when the head participant has concluded the CAT to be a failure.
(6) rollback notice (RO): This message is sent to the head participant of a conversation (say X) when a participant process learns that the conversation (say Y) which it participated in earlier has become a failure.
(Note: In fact, the PV message is not necessary. However, it is believed that the PV message is helpful in implementing and testing the system.)

Figures 6 and 7 show two different cases of execution under N-3. In both cases, Process A is the head participant of CONV1 and Process B is the head participant of CONV2. In the first case (shown in Figure 6), Processes B and C complete CONV2 with no failure. However, CONV2 is not completely validated until CONV1 is since Process B participated in CONV1. In the second case (shown in Figure 7), process A fails at (4) after Processes B and C completed CONV2. This results in CONV2 being nullified since Process B has to roll back to the recovery line of CONV1.



History	Message	Conversation status CONV1	CONV2
(1) B passes CAT of CONV1	B-->A : CV (CONV1)	og	--
(2) C passes CAT of CONV2	C-->B : CV (CONV2)	og	og
(3) B passes CAT of CONV2		og	pv
(4) A passes CAT of CONV1	A-->B : GS (CONV1)	cv	pv
	B-->C : GS (CONV2)	cv	cv

Figure 6. An example of asynchronous conversation (I).



History	Message	Conversation status CONV1	CONV2
(1) B passes CAT of CONV1	B-->A : CV (CONV1)	og	--
(2) C passes CAT of CONV2	C-->B : CV (CONV2)	og	og
(3) B passes CAT of CONV2		og	pv
(4) A fails CAT of CONV1	A-->B : FA (CONV1)	sf	pv
	B-->C : RO (CONV2)	sf	nu

Figure 7. An example of asynchronous conversation (II).

Suppose a process has already participated in conversation X and conversation Y in sequence and sent the PV message on exit from Y. When the process learns later that conversation X has become completely validated the process sends the CV message to the head participant of conversation Y. It also removes the corresponding table from the linked list. If conversation X turns out to be a failure, then the process sends the RO message to the head participant of conversation Y, and then rolls back to retry conversation X with the next alternate try block. When the execution of conversation Y has to be nullified due to the failure of conversation X, processes are allowed to use their primary try blocks when they reenter conversation Y.

4.6 Approach N-4 for execution of NLRB's: Asynchronous exit and decentralized CAT execution

This execution approach is almost the same as the previous case. Since there is no head participant under the decentralized CAT approach, the result of the global acceptance test is not broadcast. Consequently, GS (global success notice) and GF (global failure notice) messages are not required.

4.7 Discussion

As mentioned earlier the asynchronous exit approach increases the performance at the cost of implementation complexity. The process has to keep the history information until the conversation is completely validated. Moreover, in order to support fast rollback, it is necessary to incorporate the interrupt mechanism in the system. That is, when the IO handler receives the failure or rollback message it immediately interrupts the corresponding process to minimize the waste of time. Therefore, one- or two-conversation lookahead seems the most practical in a wide range of applications. (Also, the analytic results show that the incremental performance gain by allowing two-conversation lookaheed is much less than that obtained by allowing one-conversation lookahead.)

5. Implementation of the ADT-Conversation Scheme in LCN Systems

5.1 ADT-Conversation scheme

The Abstract Data Type (ADT-) Conversation scheme was proposed to remedy the shortcomings of the NLRB scheme [Kim82], which stems from the scattered appearance of the constituent RB's of a conversation structure in the program text. In the ADT-Conversation scheme, the conversation construct is structured in the form of an abstract data type.

A possible syntactic structure is shown in Figure 8. The participant enters a conversation by calling a procedure, say CONV.PROCA, of which the name and the formal parameters are listed in the participant area. The role of CO (conversation object) in the figure is to facilitate interprocess communication within the associated conversation. In other words, CO is accessible only within the conversation. This prevents information smuggling by a process participating in the conversation.

Basically, there are two options in executing the CAT. One is to decompose the CAT into segments, each executed by a different participating process. The other is to have one process execute the entire CAT after all the participating processes have completed their executions of conversation tasks.

Once the code for the CAT is decomposed (i.e., the first option is adopted), the global acceptance test is done with either the decentralized or semi-centralized approach. The implementation strategies for these cases should be the same as those discussed in the previous section (i.e., NLRB scheme cases). Therefore, in the following subsections 5.3 and 5.4 strategies for implementing the ADT-Conversation scheme with centralized CAT approach in combination with either the synchronous or the asynchronous exit approach are discussed.

5.2 Syntax adopted and message format

The syntax given in Figure 8 can be incorporated into most of the target implementation languages, although some variations are possible. The ADT-Conversation incorporated into Path Pascal [Kol80] was implemented by the authors [Kim85]. The same message format given in the previous section (Figure 4) is used here again.

5.3 Approach A-1 for execution of ADT-Conversations: Synchronous exit and centralized CAT execution

In the centralized CAT approach, only one participant executes the acceptance test and broadcasts the result. It is possible to designate the participant that executes the CAT in two ways: statically and dynamically. With static designation, the predetermined head participant executes the CAT. With dynamic designation, on the other hand, the last participant that completes the conversation task executes the CAT. Under the synchronous exit approach the effect of this choice is relatively insignificant since all other participants should wait until the last participant completes the conversation task.

Only two types of CAT result messages are needed since no local acceptance test is done by the participant.
(1) global success notice (GS): This message is broadcast when the CAT has succeeded.
(2) global failure notice (GF): This message is broadcast when the CAT has failed.

5.4 Approach A-2 for execution of ADT-Conversations: Asynchronous exit and centralized CAT execution

The dynamic designation approach in executing the CAT, in general, performs better than the static designation approach under asynchronous exit. This is because under the static approach some extra work is required for the last participant to inform the head participant of completion of the conversation task. Nevertheless, the static approach seems more practical in the sense that it is easier to debug and monitor the system.

Even if the CAT has resulted in a pass, the conversation is not completely validated unless all the participants are irrevocable. Therefore, the global success

```
type C = conversation

<const & type declaration>   "can declare nested conversation"

participants
        PROCA (... "formal parameters" ...);
        PROCB (  ..................................  );
        ............
var
    CO: c-object-type;        "conversation object declaration"
    ..........
<CAT function declaration>   "conversation acceptance test"
<procedures & functions declaration>

ensure CAT
by begin                      "primary interacting session"
   PROCA: begin ..... end;
   PROCB: ......
   ............
 end;
eliseby begin                 "alternate interacting session"
   PROCA: begin ..... end;
   PROCB:   .........
   ..............
 end;
.............
elseerror

endconversation;
```

Figure 8. ADT-Conversation.

notice is deferred if there is at least one revocable participant. Three types of CAT result messages are needed.
(1) global success notice (GS): This message is broadcast when the CAT has suceeded and all the participants are irrevocable.
(2) global failure notice (GF): This message is broadcast when the CAT has failed.
(3) rollback notice (RO): This message is sent to the head participant of a conversation (say X) when a participant process learns that the conversation (say Y) which it participated earlier has become a failure.

5.5 Discussion

As pointed out in [Kim82], the ADT-Conversation scheme has a number of advantages over the NLRB scheme. Among others, (1) each interacting session is presented as a single unit in the program text and thus easier to read and (2) it is not necessary to decompose the CAT into distributed routines of which collective effect is generally harder to comprehend. Also, all six implementation approaches (combinations of three different CAT approaches and both synchronous and asynchronous exit cases) are applicable. However, it seems natural to have the centralized CAT for the ADT-Conversation scheme because decomposition of CAT requires some extra work.

Under the centralized CAT approach, in contrast to the decentralized or semi-centralized CAT approach, no

local acceptance test is done by each participant. This possibly degrades detection and recovery performance because the CAT is not evaluated until all the participants complete the conversation tasks. Under the decentralized or semi-centralized CAT approach, it is possible to have participants abandon the conversation tasks if one of the participants has failed in its local acceptance test. This reduces unnecessary computation time although the amount of gain is highly application-dependent.

6. Summary

This paper presented several different approaches for implementing the conversations scheme in loosely coupled DCS's. Two different exit control strategies, one in synchronous manner and the other in asynchronous manner, and three different approaches to execution of CAT, centralized, decentralized, and semi-centralized, have been examined and compared in terms of system performance and implementation cost. Since each approach has merits and deficiencies, it is hard to say that one approach is simply better than another. Moreover, the implementation strategy should be carefully chosen based on the characteristics of the application system such as network topology, communication cost, etc.

However, it seems that the asynchronous exit approach is generally better than the synchronous exit approach. The former provides a higher performance during error-free execution than the latter. If the NLRB structuring approach is used then the semi-centralized CAT approach or the decentralized CAT approach is more attractive than the centralized approach. On the other hand, if the ADT-conversation structuring approach is used then the selection of a good strategy for execution of a CAT depends on whether one can afford the effort required to decompose the CAT into distributed acceptance tests. If so, the semi-centralized or decentralized approach is more attractive and otherwise the centralized CAT execution is the only choice.

Incorporation of the timeout capability into the conversation scheme is an area to be studied [Hec76]. The crash of a participant (or a node) can result in the lockup of several other nodes if the timeout mechanism is not used. Since each participant enters the conversation asynchronously, the timeout period is an important design parameter, and an effective technique for determination of a proper timeout period needs to be developed. Integration of the conversation scheme and other established fault tolerance schemes [Bha87, Kim84,Toy87] is also an important area for future research.

The authors have obtained analytic results on system performance under various workload conditions and have reported some of the results in [Kim86,Kim88]. The result shows that the asynchronous exit reduces the synchronization overhead of the conversation scheme to a significant extent (only with one- or two-conversation lookahead). However, in order to obtain "real" data on implementation cost and system performance, further experimental work is necessary. Testbed-based evaluation [Chu87] of the proposed approaches in the context of a real world application is regarded as a highly worthwhile research topic.

7. References

[Bha87] Bhargava, B., editor, 'Concurrency and Reliability in Distributed Systems', Van Nostrand and Reinhold, 1987.

[Chu87] Chu, W.W., Kim, K.H., and McDonald, W.C., "Testbed-Based Evaluation of Design Techniques for Fault-Tolerant Real-Time Distributed Computer Systems", Proc. of the IEEE, Vol. 75, No. 5, May 1987, pp.649-667.

[Cam83] Campbell, R.H., Anderson, T., and Randell, B., "Practical Fault Tolerant Software for Asynchronous Systems", Proc. SAFECOM 83, Cambridge, Oct. 1983, pp.59-65.

[Gre85] Gregory, S.T. and Knight, J.C., "A New Linguistic Approach to Backward Error Recovery", Proc. FTCS-15, 1985, pp.404-409.

[Hec76] Hecht, H., "Fault-Tolerant Software for Real-Time Applications", Computing Surveys, Dec. 1976, pp.391-407.

[Kim76] Kim, K.H., Russell, D.L., and Jenson, M.J., "Language Tools for Fault-Tolerant Programming", PETP-1, Electronic Sciences Lab., USC, Nov. 1976.

[Kim82] Kim, K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitor", IEEE Trans. on Software Eng., Vol. SE-8, No. 3, May 1982, pp.189-197.

[Kim84] Kim, K.H., "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults", Proc. the 4th Int'l Conf. on DCS, May 1984, pp.526-532.

[Kim85] Kim, K.H., Yang, S.M., and Kim, M.H., "Implementation of Concurrent Programming Language Facilities Supporting Conversation Structuring", Proc. COMPSAC 85, Oct. 1985, pp.445-453.

[Kim86] Kim, K.H., Heu, S., and Yang, S.M., "An Analysis of the Execution Overhead Inherent in the Conversation Scheme", Proc. 5th Symp. on Rel. in Distributed Software and Database Systems (5SRDS), Jan. 1986, pp.159-168.

[Kim88] Kim, K.H. and Yang,S.M.,"An Analysis of the Performance Impacts of Lookahead Execution in the Conversation Scheme," Proc. IEEE Computer Society's 7th Symp. on Reliable Distributed Systems, Oct. 1988, pp.71-81.

[Kol80] Kolstad, R.B. and Campbell, R.H., Path Pascal User Manual, Univ.of Illinois at Champaign-Urbana, Jan. 1980.

[RAN75] Randell, B., "System Structure for Software Fault Tolerance", IEEE Trans. on Software Eng., June 1975, pp.220-232.

[RUS79] Russell, D.L. and Tiedeman, M.J., "Multiprocess Recovery using Conversations", Proc. FTCS-9, 1979, pp.106-109.

[Toy87] Toy, W.N., "Fault-Tolerant Computing", A chapter in Advances in Computers, Vol. 26, Academic Press, 1987, pp.201-279.

Appendix A.VII

Implementation of the Conversation Scheme in Message-Based Distributed Computer Systems

S. M. Yang & K.H. (Kane) Kim

October 1990

Technical Report
UCI-ECE-90-12

S. M. Yang
Department of Computer Science
University of Texas at Arlington
P.O.Box 19015
Arlington, TX 76019
(817) 273-3629
Yang@Evax.Utarl.Edu

K. H. (Kane) Kim
Department of Electrical & Computer Engineering
University of California
Irvine, California 92717
(714) 856-5542 (office), 856-4076 (FAX)
Internet: Kane@Ics.Uci.Edu

IMPLEMENTATION OF THE CONVERSATION SCHEME IN MESSAGE-BASED DISTRIBUTED COMPUTER SYSTEMS

ABSTRACT

This paper discusses several different approaches for implementing conversations in message-based distributed computer systems (DCS's). Important implementation factors to be considered include the control of exits of processes upon completion of their conversation tasks and the approach to execution of the conversation acceptance test. Two different exit control strategies, one in a synchronous manner and the other in an asynchronous manner, and three different approaches to execution of the conversation acceptance test, centralized, decentralized, and semi-centralized, are examined and compared in terms of system performance and implementation cost. A new efficient approach to run-time management of recovery information based on an extension of the recovery cache scheme is also discussed. The effectiveness of these execution approaches also depends on the way conversations are structured initially by program designers. Therefore, the two major types of conversation structures, Name-Linked Recovery Block (NLRB) and Abstract Data Type (ADT) Conversations, are examined to analyze which execution approaches are the most efficient for each conversation structure. These results provide useful guidelines for implementing conversations in message-based DCS's. As a case study, an unmanned vehicle system is used to illustrate how the identified approaches to implementation of the conversation scheme can be used in a realistic real-time application.

Index Terms: fault-tolerant cooperating processes, conversation, acceptance test, recovery line, message-based distributed system, lookahead, real-time control system.

1. Introduction

Since the concept of conversation structuring was introduced in an abstract form as an approach to facilitating cooperative recovery in systems of interacting processes [Ran75], continuous research efforts have been invested to convert the concept into a practical technology. For example, several mechanized structuring schemes containing practical language syntax and associated precise semantics have been formulated [Cam83,Gre85,Kim82,Rus79]. Subsequently, some practical language processing systems have been produced [Kim85], a "recovery metaprogram" has been proposed for efficient design of conversations [Oza88] and the execution costs of conversations have been studied [Kim89]. Conversation design schemes using Petri Nets have also been studied [Tyr86, Wu89]. In [Man89] an attempt is made to utilize some well-known object replication techniques in design of communicating processes with conversation.

However, the experimental study of the scheme has been scarce and has not yet progressed to the point of producing concrete results. Moreover, techniques for efficient implementation, especially in distributed computer systems (DCS's), have not been much studied either. This paper presents issues in implementing conversations in DCS's together with some efficient implementation approaches. The following three cost factors should be carefully considered in such implementation.

The first factor is the cost of communication between remote processes. In selecting an efficient implementation strategy of the conversation scheme, this interprocess communication cost plays an important role as will be elaborated in this paper. The communication cost is primarily determined by the network structure adopted, i.e., network topology, communication medium, and protocol used. In this paper "message-based DCS's" in which computing nodes communicate with each other via message passing are considered.

The second factor is the cost of synchronizing processes at the exit of a conversation. As shown in [Kim86], this cost can have significant impact on the performance of the conversation scheme. To reduce this cost, the approach of conversation lookahead, i.e., to allow asynchronous exits from the conversation, was studied in [Kim89] for its potential performance impacts based on an analytic model. The conversation scheme extended with the lookahead capability is called the asynchronously exited conversation scheme whereas the conversation scheme with no lookahead is called the synchronously exited conversation scheme. Under the synchronous exited conversation scheme, all processes are synchronized at the end of each conversation, i.e., no process is allowed to exit from the conversation earlier than other processes even if the process has passed its

acceptance test. On the other hand, under the asynchronously exited conversation scheme processes are allowed to exit from the conversation as they finish their tasks, but the exiting processes maintain the capabilities for their rollback to the beginning of the conversation until all processes have passed their acceptance tests and exited from the conversation.

The third factor is the cost of designing and executing the conversation acceptance test (CAT). In a single-node/multiprocess system, the choice between the centralized CAT (i.e., one participant takes care of the non-local acceptance test routine) and the decentralized CAT (i.e., each participant has its own acceptance test routine), has relatively insignificant effect on the system performance. However, in a message-based DCS, the CAT choice affects the system performance to a significant extent. In this paper, three different approaches to design and execution of CAT's, centralized, decentralized, and semi-centralized, are discussed.

In the next section, approaches to implementation of synchronous and asynchronous exit control strategies are introduced and pros and cons of each approach are discussed. An efficient approach to run-time management of recovery information based on an extension of the recovery cache scheme is also discussed. In Section 3, three different CAT execution approaches are presented. The effectiveness of different execution approaches also depends on the way conversations are structured by program designers. The cases of using two different basic types of conversation structures, Name-Linked Recovery Block (NLRB) and Abstract Data Type (ADT) Conversation, are examined in Sections 4 and 5 respectively, in order to analyze the effectiveness of the execution approaches discussed earlier. In spite of the passage of a considerable amount of time since the publication of the first paper [Ran75] that proposed the concept of conversation structuring, practical illustrations have been scarce. In Section 6, an unmanned vehicle system is used to illustrate how the approaches discussed in earlier sections for implementation of the conversation scheme can be used in a realistic real-time application. Section 7 is the summary section.

2. Synchronously Exited and Asynchronously Exited Conversations

This section starts with a brief description of the conversation structure. Then the characteristics of synchronously exited and asynchronously exited conversations are discussed. These two approaches are quite different in terms of their impacts on system performance and complexity. A brief comparison is made at the end of this section.

2.1 Basic logical structure of the conversation

The conversation is a two-dimensional enclosure of recoverable activities of multiple interacting processes, in short, a recoverable interacting session [Kim82, Ran75]. It creates a "boundary" which process interactions may not cross. The boundary of a conversation consists of a recovery line, a test line and the walls defining exclusive membership as shown in Figure 1. Each participant process contains one or more try blocks designed to produce the same or similar computational results as well as an acceptance test which is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A recovery line is a coordinated set of the recovery points of interacting processes that are established (possibly at different times) before interactions begin. A test line is a correlated set of the acceptance tests of interacting processes.

A conversation is successful only if all the interacting processes pass their acceptance tests forming the test line. Therefore, the participants are allowed to leave the conversation when all the participants have passed their acceptance tests. If any of the acceptance test fails, all the processes roll back to the recovery line and retry with their alternate try blocks. These alternate try blocks collectively define an alternate interacting session (AIS) whereas the set of primary try blocks executed first after the processes enter the conversation define the primary interacting session (PIS). A process that has executed its try block and passed its acceptance test is said to have finished its conversation task. A process which is inside a conversation cannot interact with a process which is not in the conversation. Conversations must be strictly nested in two dimensions. That is, when conversation C.nest is nested within conversation C, the set of processes that participate in nested conversation C.nest must be a subset of the processes that participate in C, the entire recovery line of C.nest must be established after the entire recovery line of C, and the entire test line of C.nest must be set before the entire test line of C.

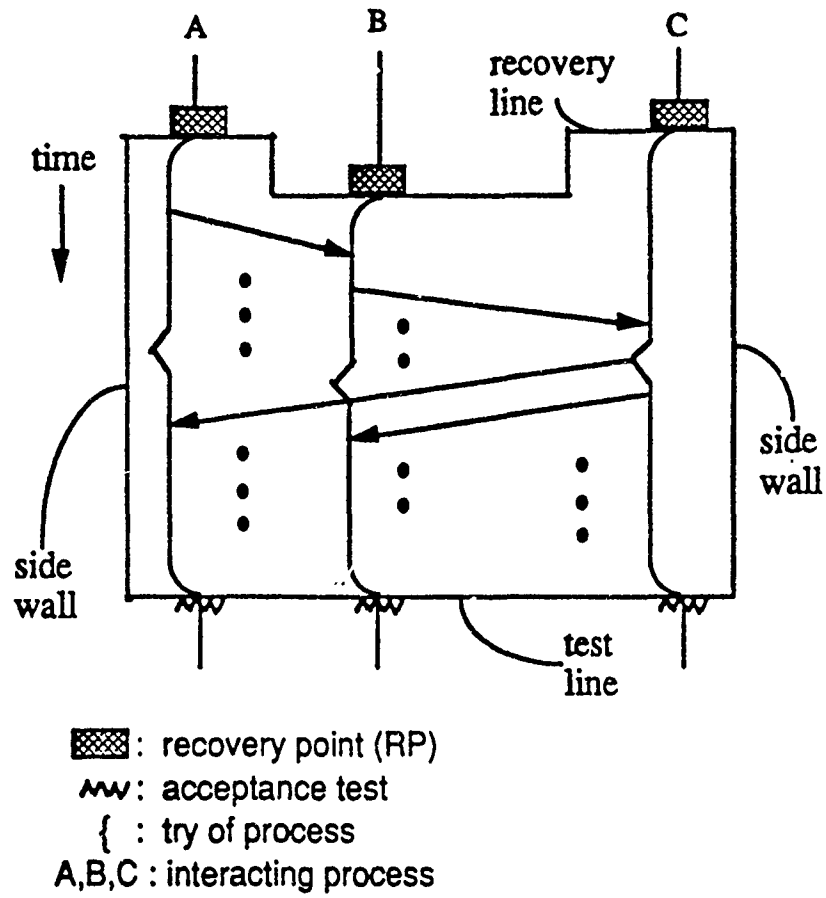


Figure 1. Conversation (adapted from [Kim82]).

2.2 Exit control

In the basic conversation scheme sketched above, processes enter a conversation asynchronously but synchronize themselves before exiting from it. The synchronization considered here is of a special kind specifically required by the conversation scheme and thus different from the application-dependent synchronization required between cooperating processes. The synchronization can add significantly to the time cost of the conversation scheme. A fundamental approach to reducing the synchronization overhead is the lookahead. Under the lookahead approach each participant process leaves the conversation as soon as it passes its own acceptance test. It does so with the awareness of the possibility that another participant may execute an acceptance test later and fail in the test thus making it necessary for the former to roll back to the recovery line of the conversation. Therefore, the lookahead approach here is an opti-

validated if CONV has been validated and there is at least one conversation which is not only logically before CONV but also in the on-going state.

For example, in Figure 3 conversations CONV1, CONV2 and CONV3 are logically before CONV4. Therefore, CONV4 cannot be completely validated (even if it has been validated) until all of its logically earlier conversations CONV1, CONV2 and CONV3 become completely validated. Also, CONV5 is logically before CONV6. However, CONV5 and CONV6 have no ordering relationship with any other earlier entered conversations (i.e., CONV1, CONV2, CONV3, and CONV4 in this example). Therefore, CONV6 becomes completely validated if it has been validated and CONV5 has been completely validated.

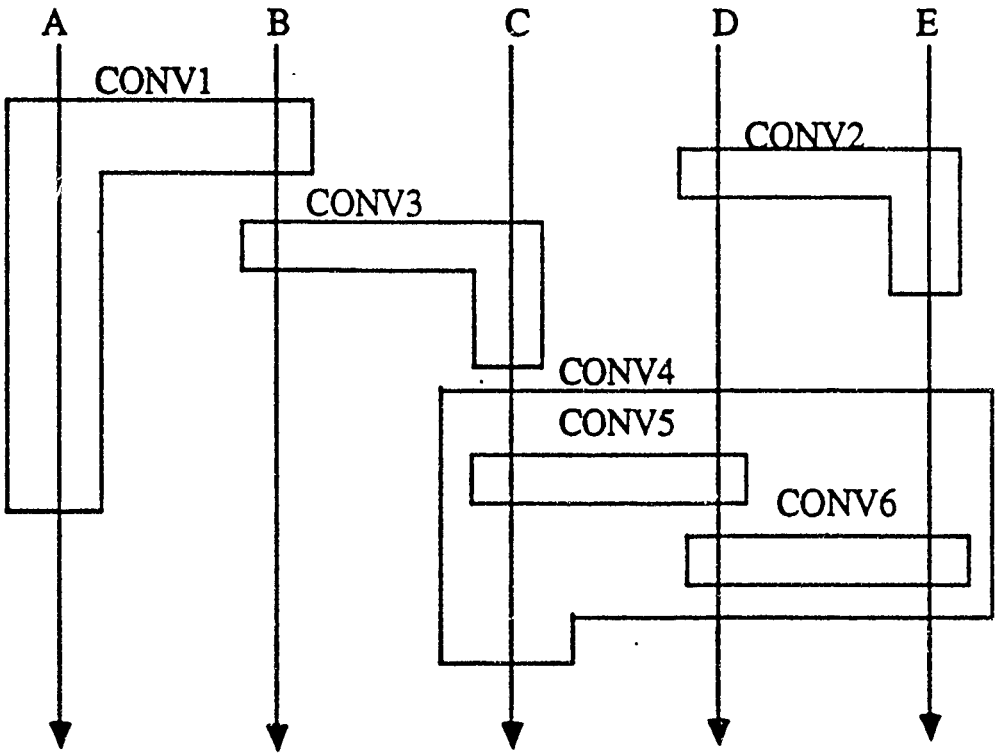


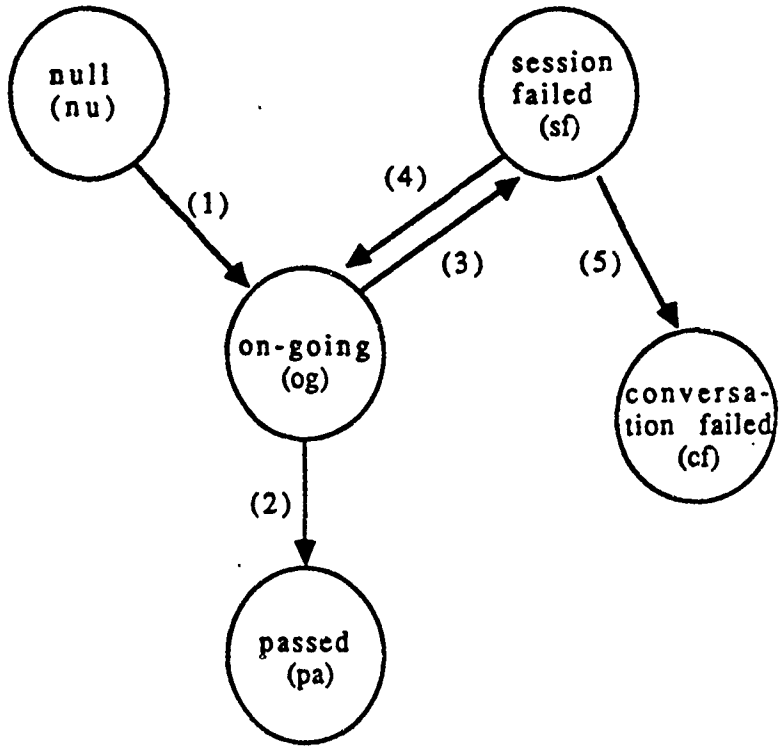
Figure 3. An Example of asynchronously exited conversation.

An asynchronously exited conversation may, therefore, transit among the following six different states as shown in Figure 4.

(S1) *null*: None of the participants have entered into the conversation.

(S2) *on-going*: There is at least one participant which is executing the conversation task.

(S3) *partially validated*:



(1) Start execution of the primary interacting session.
(2) All the participants have passed their acceptance tests.
(3) A participant has failed in its acceptance test.
(4) Start execution of the next alternate interacting session.
(5) A conversation has failed due to the failures of all interacting sessions.

Figure 2. Change in the state of a synchronously exited conversation.

Definition 1: A conversation is validated if all the participants of the conversation have passed their acceptance tests.

Definition 2: When a process exits from a conversation and enters another conversation, the latter (the former) conversation is said to be logically after (before) the former (the latter) conversation. On the other hand, when a process in a conversation enters another (nested) conversation (without exiting from the former conversation), the newly entered nested conversation has no logical ordering relationship with any order earlier entered conversations.

Definition 3: A conversation CONV is completely validated (1) if CONV has been validated and there is no other conversation which is not only logically before CONV but also in the on-going state, or (2) if CONV has been validated and all other conversations which are logically before CONV have been completely validated.

Definition 4: A conversation CONV is partially validated if CONV is neither in the on-going state nor in the completely validated state. To be more specific, a conversation CONV is partially

validated if CONV has been validated and there is at least one conversation which is not only logically before CONV but also in the on-going state.

For example, in Figure 3 conversations CONV1, CONV2 and CONV3 are logically before CONV4. Therefore, CONV4 cannot be completely validated (even if it has been validated) until all of its logically earlier conversations CONV1, CONV2 and CONV3 become completely validated. Also, CONV5 is logically before CONV6. However, CONV5 and CONV6 have no ordering relationship with any other earlier entered conversations (i.e., CONV1, CONV2, CONV3, and CONV4 in this example). Therefore, CONV6 becomes completely validated if it has been validated and CONV5 has been completely validated.

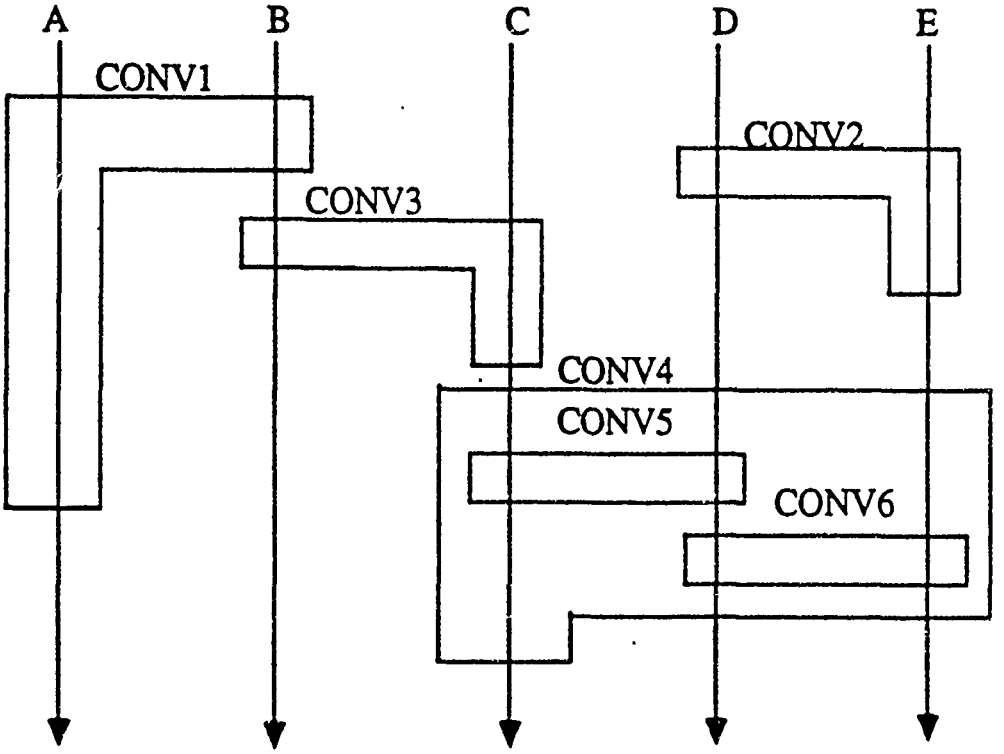


Figure 3. An Example of asynchronously exited conversation.

An asynchronously exited conversation may, therefore, transit among the following six different states as shown in Figure 4.

(S1) *null*: None of the participants have entered into the conversation.

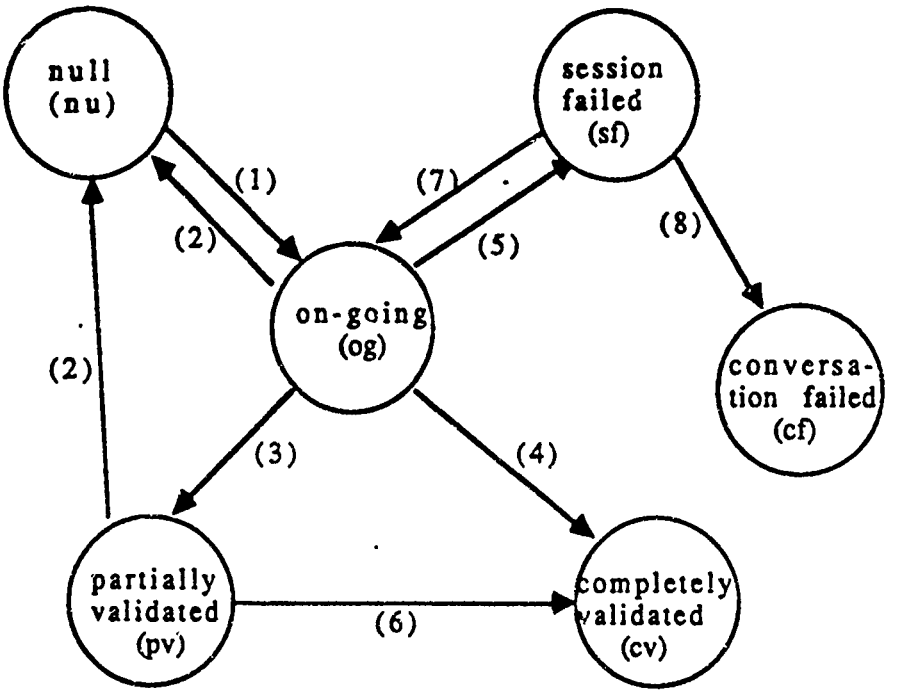
(S2) *on-going*: There is at least one participant which is executing the conversation task.

(S3) *partially validated*:

(S4) *completely validated*:

(S5) *session failed*: At least one participant has failed in its acceptance test.

(S6) *conversation failed*: The conversation has failed due to the failures of all interacting sessions.

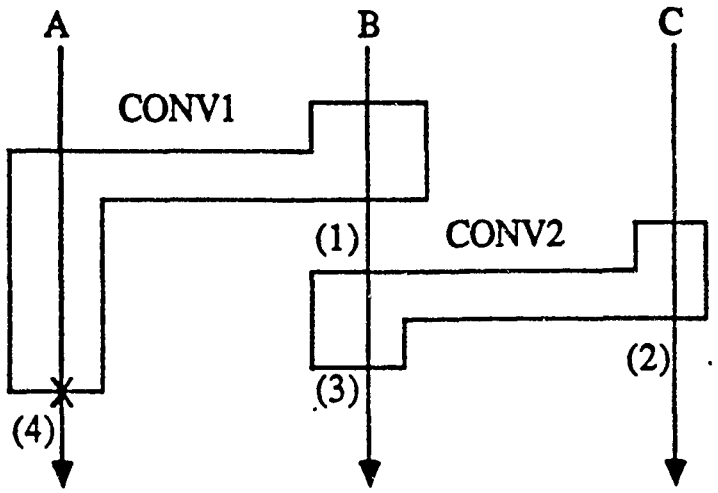


(1) Start execution of the primary interacting session.
(2) Conversation execution is nullified due to the rollback of a participant to an older conversation.
(3) All the participants have passed their acceptance tests but there is at least one logically earlier conversation which is in the on-going state.
(4) All the participants have passed their acceptance tests and there is no logically earlier conversation which is in the on-going state.
(5) A participant has failed in its acceptance test.
(6) All logically earlier conversations have become completely validated.
(7) Start execution of the next alternate interacting session.
(8) A conversation has failed due to the failures of all interacting sessions.

Figure 4. Change in the state of an asynchronously exited conversation.

In Figure 4 the transition (2) means that the on-going or partially validated conversation is nullified when one of the participants must roll back to an older conversation. For example, in Figure 5 Processes A and B participate in CONV1. Then Process B participate in CONV2. Suppose Processes A fails at (4) after Processes B and C completed CONV2 (that is, CONV2 has been partially validated). This results in CONV2 being nullified since Process B has to roll back to

the recovery line of CONV1. If a retry of the older conversation (i.e., CONV1) is successful, Process B and C may again execute the conversation (i.e., CONV2) with their primary try blocks (transition (1) in Figure 4).



History	Message	Conversation State CONV1	CONV2
(1) B passes CAT of CONV1	B-->A: "pass"	og	--
(2) C passes CAT of CONV2	C-->B: "pass"	og	og
(3) B passes CAT of CONV2	B-->C: "pass"	og	pv
(4) A fails CAT of CONV1	A-->B:."fail"	sf	pv
	B-->C: "nullify"	sf	nu

Figure 5. An example of an asynchronously exited conversation.

Intuitively, the system performance would increase if asynchronous exits of the participants are allowed. The performance improvement would be particularly conspicuous when the acceptance test failure probability is very low (making the rollback infrequent) and the number of participants is large (making the synchronization overhead substantial). A previous analytic study on the performance of the conversation scheme based on a queueing network model [Kim89] confirmed this property. It also showed that the performance of a system would be significantly affected by the synchronization required of the processes in exiting from a conversation, not by the failure probability of each process. For example, suppose a process is allowed to make "one conversation lookahead", which means that a process can continue lookahead as long as it has not exited from more than one unfinished (i.e., on-going or partially validated) conversation. The analytic study showed that by allowing one conversation lookahead the performance could increase about 46% when the number of participants is six, the failure probability is almost zero, and the

amount of computation that a process performs inside a conversation is on the average about 10% of the total computation the process performs. On the other hand, the performance would decrease only 2% when the failure probability increases from zero to 0.05 (which is higher than that of most practical systems) and other parameters including synchronous exit remain unchanged. (Further details are referred to [Kim86, Kim89].)

2.4 Management of recovery information

Under the asynchronous exit approach, the information needed to roll back, e.g., rollback point, values of the non-local variables which have been changed after the process entered the conversation, etc., should be kept until the conversation is completely validated. In order to keep this information each process maintains a table called the "conversation record" for each unfinished conversation. The entries of the conversation record are shown in Figure 6. The table is created when the process initially enters the conversation and removed when the conversation becomes completely validated.

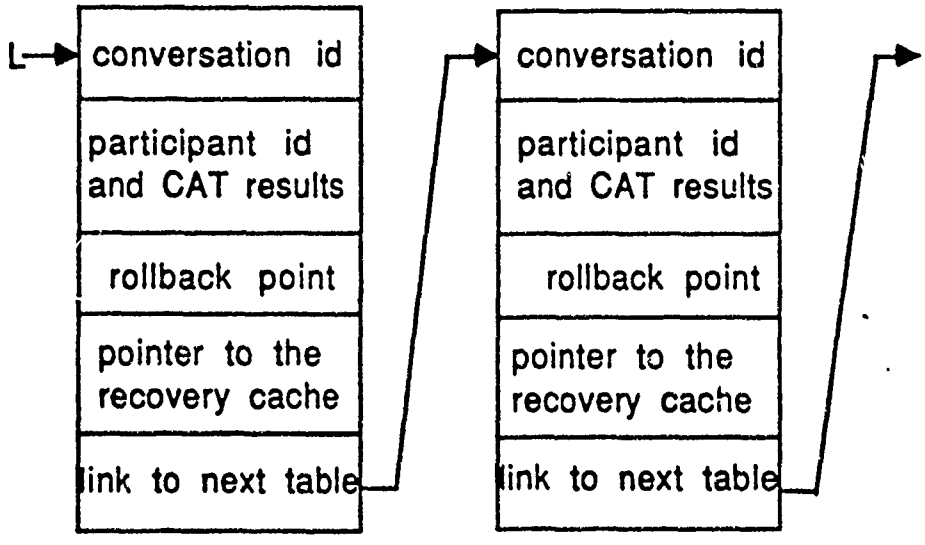


Figure 6. Conversation record.

The conversation records are linked hierarchically as they are created. All partially validated or on-going level-0 conversations (i.e., conversations with no parent conversations) are linked linearly. The conversation record of a nested conversation (i.e., level-1 conversation which is immediately nested within a level-0 conversation or lower level conversation) is linked under its parent record. Therefore, records of various unfinished conversations (including partially

validated conversations) are in general related in the form of a tree. Figure 7 illustrates a tree of conversation records created in support of nested conversations. Figure 7.b shows how the conversation record is managed in Process B under the following scenario: (In Figure 7.b a solid box represents a conversation record for an on-going conversation, a dotted box for a partially validated conversation, and a shaded box for a completely validated conversation.)

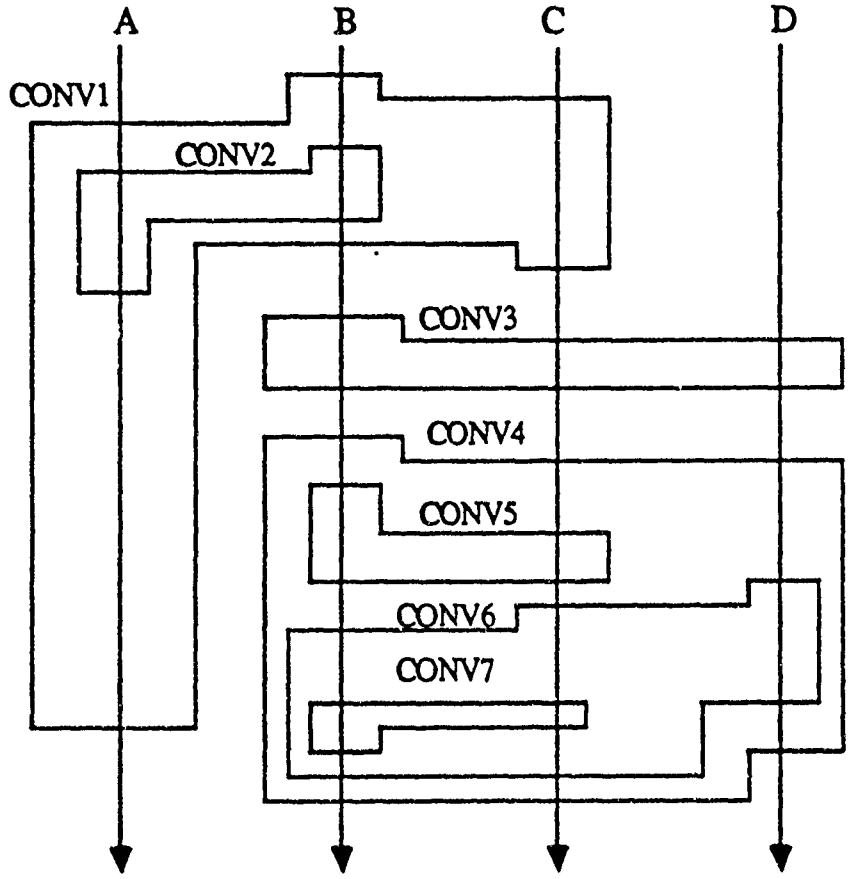


Figure 7.a. An illustration of nested conversations.

(a) Process B is executing a level-1 conversation CONV2, i.e., CONV1 and CONV2 are on-going conversations.

(b) Process B completed CONV2 and then CONV1 while Process A has completed CONV2. Therefore, CONV2 has been completed validated within CONV1.

(c) Process B has entered another level-0 conversation CONV3 while Process A is still executing CONV1.

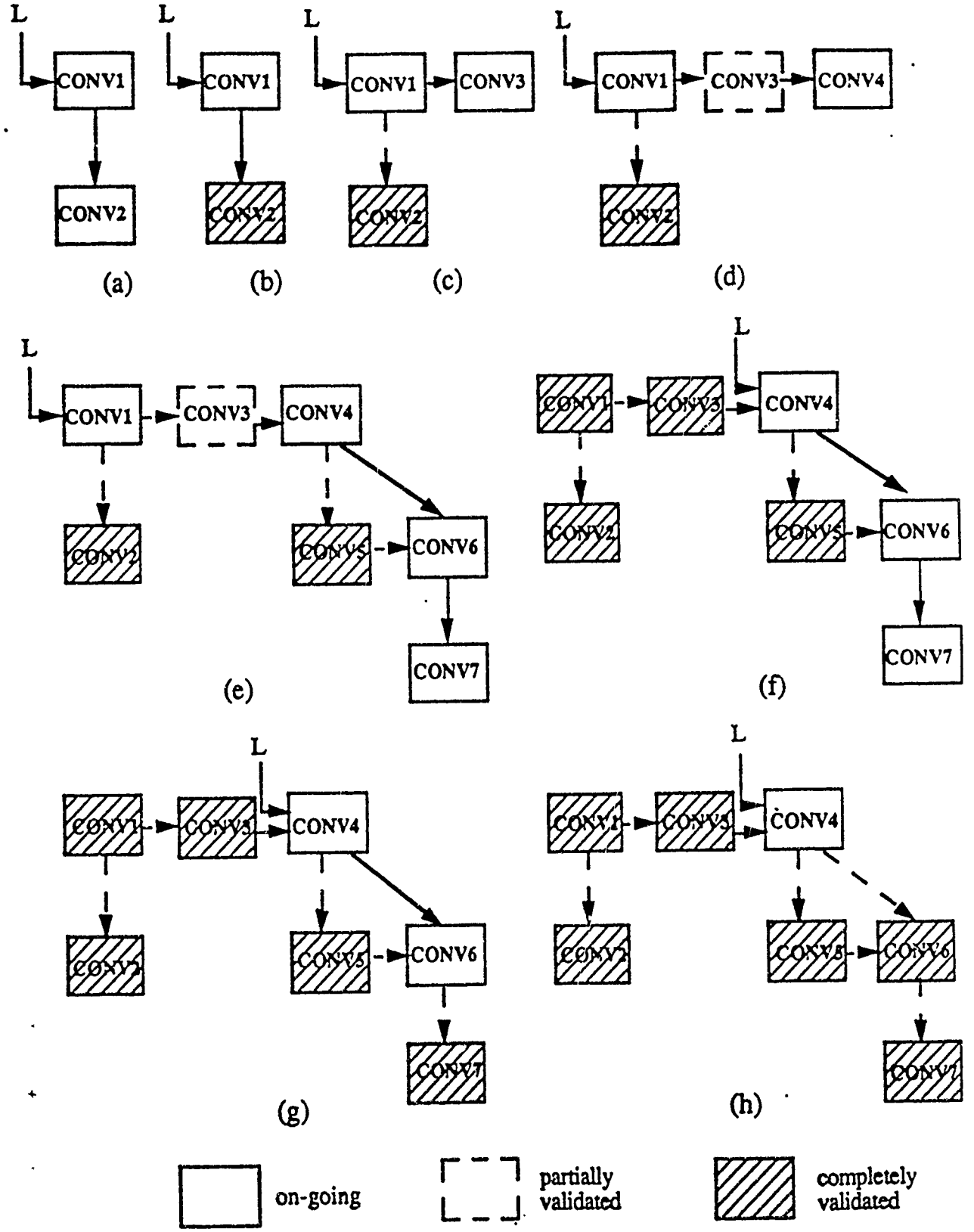


Figure 7.b. Conversation record structure for nested conversations.

(d) Processes B, C and D completed CONV3 and have entered another level-0 conversation CONV4. CONV3 has been partially validated since CONV1 which is logically before CONV3 is still on-going.
(e) Processes B and C entered and completed CONV5, have entered CONV6, and then CONV7 (a level-2 conversation). Therefore, CONV5 was completely validated within CONV4 and CONV6 and CONV7 are on-going. CONV1 has not been completely validated yet, because Process A is still in CONV1.
(f) Process A has finally completed CONV1. CONV3 becomes completely validated since its logically earlier conversation, CONV1, has been completely validated. The conversation records of CONV1 and CONV3 are removed from the link.
(g) Processes B and C have completed CONV7. Therefore, CONV7 is completely validated within CONV6.
(h) Processes B, C and D have completed CONV6. CONV4 is now the only on-going conversation.

Another implementation consideration is how to keep the values of non-local variables, i.e., how to save the computation state after the process entered conversation. An approach, which is an extension of the recovery cache scheme developed in [Hor74, And76] to facilitate efficient execution of recovery blocks, is discussed here. Under the recovery cache scheme a special run-time stack, named a cache stack, is used to save the old values of non-local variables. When a non-local variable is assigned a value for the first time after a recovery point has gone past, the existing value together with the variable name is saved into the top region of the cache stack. Therefore, using the values in the top region of the cache stack, the process can roll back to the recent recovery point. This also means that whenever a non-local variable is to be assigned a value, it must be checked whether it is the first assignment after a recovery point has gone past. In the rest of this subsection we briefly discuss how the recovery cache can be extended to facilitate conversation execution.

The cache stack of an ordinary stack structure is not sufficient for facilitating asynchronously exited conversations. This is because the stack needs to shrink in two directions: (1) to shrink from the top when a process has to roll back, which requires restoration of the modified non-local variables, and (2) to shrink from the bottom when a conversation has been completely validated, which requires discard of the old values. In normal situation (i.e., when there is no fault) the stack grows in one direction (when the old values of non-local variables are saved in the stack) and shrinks from the other direction (when the saved old values are discarded from the stack).

Therefore, this type of stack can be implemented as a "circular" stack, i.e., a stack with two pointers, one for the "stack_top" and one for "stack_base".

When the process enters the conversation the current stack_top (which becomes the stack_base of this conversation) is saved in the conversation record. The entry for saving this pointer is shown as "pointer to the recovery cache" in Figure 6. When the process has to roll back to the conversation which has failed or been nullified, the values of non-local variables saved in the region defined by the current stack_top and the stack-base saved in the conversation record are restored. If the same variables appear more than once then the oldest values should be restored.

In the case of nested conversations (i.e., level-1 or lower level conversations), even after a nested conversation is completely validated within its parent conversation, the values of the non-local variables which have been modified within the nested conversation should be kept until its parent conversation becomes completely validated. In other words, the segment which belongs to the nested conversation in the recovery cache stack gets merged into the segment belonging to the parent conversation. When two stack segments get merged into one the same variables may appear twice, i.e., one in the segment of the nested conversation and one in the segment of the parent conversation. There are two approaches to handling this problem. First, we discard the useless ones, i.e., the younger ones, immediately. Second, instead of discarding those immediately, we keep as they are in the stack until the whole segment is removed from the stack. If the restoration is needed (due to rollback) the younger ones are ignored. Although the first approach saves some memory space, the run-time overhead is higher than the second approach. Therefore, the second approach seems more suitable for real-time applications.

In Figure 8.a, CONV1 and CONV3 are level-0 conversations and CONV2 is a nested (level-1) conversation within CONV1. Suppose that in Process B
(1) non-local variables p, q, and s are assigned new values within CONV1;
(2) non-local variables q and r are assigned new values within CONV2;
(3) non-local variables r and t are assigned new values after completion of CONV1 and before initiation of CONV3; and
(4) non-local variables q and u are assigned new values within CONV3.

Figure 8.b shows how the recovery cache stack is managed at run-time in Process B. The stack grows in the upward direction. The left column is for variable names and the right column is for saved values.
(b1) Initially both the stack_top and stack_base point to the bottom of the stack.

(b2) Process B is about to enter CONV2.
(b3) Process B is about to leave CONV1 (assuming that CONV2 has not been completely validated.)
(b4) Process B is about to enter CONV3. (CONV2 still has not been completely validated.)
(b5) Process A has completed CONV2. Therefore, CONV2 has become completely validated within CONV1. (If Process A has failed in CONV2 then the values of q, r, s and t should be restored. Since r appears twice the older one, 4 in this example, is restored.)
(b6) Process A has completed CONV1 thereby making CONV1 completely validated. (Variables q and u have been modified within CONV3.)
(b7) Processes B and C have completed CONV3 thereby making it completely validated.

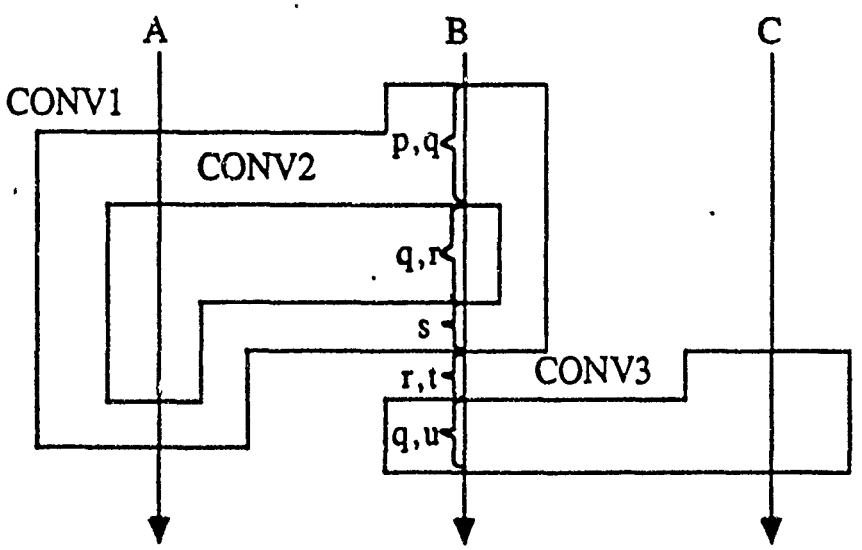


Figure 8.a. An example of an asynchronously exited conversation.

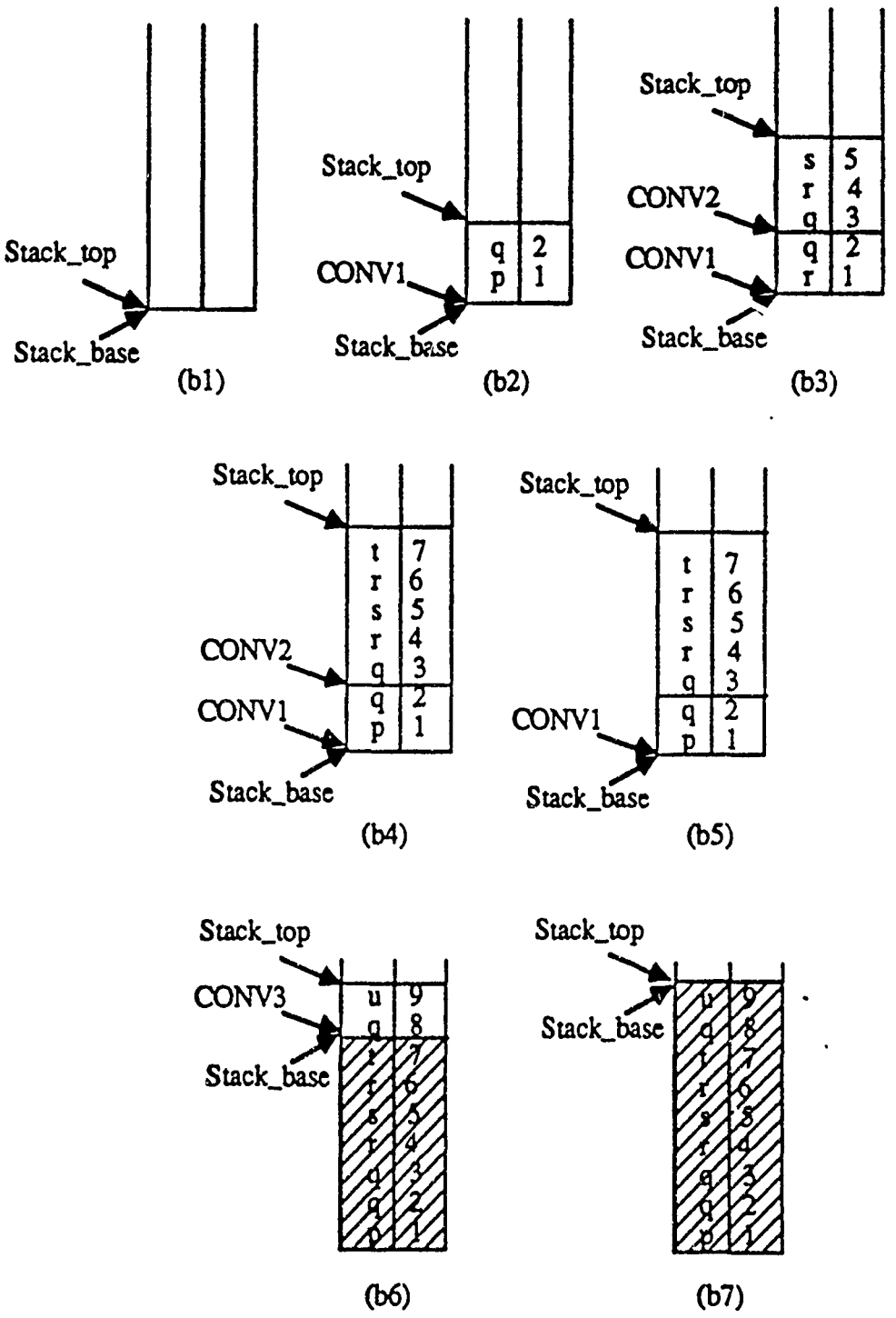


Figure 8.b. Snap shots of the recovery cache stack in Process B of Figure 8.a.

2.4 Discussion

The asynchronously exited approach increases the performance at the cost of implementation complexity. The process must keep the history information until the conversation is completely validated. As discussed in the previous section, run-time overhead due to handling of conversation records and recovery cache stacks would increase rapidly as the lookahead level increases. Moreover, in order to support fast rollback, it is necessary to incorporate the interrupt mechanism in the system. That is, when the IO handler receives a failure or rollback message, it immediately interrupts the corresponding process to minimize the waste of time. Therefore, the one- or two-conversation lookahead strategy seems the most practical in a wide range of applications. (Also, analytic study results showed that the incremental performance gain expected when changing from the one-conversation lookahead strategy to the two-conversation lookahead strategy would be relatively insignificant in comparison to the incremental gain expected from the change from the synchronous exit strategy to the one-conversation lookahead strategy [Kim89].)

3. Conversation Acceptance Test

This section describes three different major approaches to execution of the conversation acceptance test (CAT): centralized CAT, decentralized CAT, and semi-centralized CAT. Advantages and disadvantages of each approach are discussed.

3.1 Centralized CAT

In this approach only one designated participant, named "head" participant, contains the complete CAT routine. Therefore, the head participant executes the CAT when all the participants have finished their execution of try blocks and then broadcasts the CAT result to other participants. Since the code that implements the CAT is not scattered among the participant processes this approach has an advantage of not requiring the decomposition of the CAT routine. This property is valuable where the CAT is designed as a single function.

In the centralized CAT approach, it is possible to designate the head participant in two ways: statically and dynamically. With static designation, the predetermined head participant executes the CAT. With dynamic designation, on the other hand, the last participant that completes the conversation task executes the CAT. Under the synchronous exit approach the effect of a choice between static and dynamic designations is relatively insignificant since all other participants should wait until the last participant completes the conversation task. However, in the case of allowing asynchronous exits the dynamic designation approach generally performs better than the static designation approach. This is because under the static approach some extra work is required for the last participant to inform the head participant of completion of the conversation task. Nevertheless, the static approach seems more practical in the sense that it is easier to debug and monitor the system.

The total number of messages needed for CAT under this approach is N where the number of participants in the conversation is N. (In fact, this number is the same for all three CAT execution approaches as will be shown in the following sections. However, the messages communicated under the three approaches are different in length and number of destination processes.) In this approach, N-1 messages out of the N messages would be of a sizable type because the messages include the values of the variables needed for CAT. Therefore, the centralized approach may lead to relatively large communication overhead due to these sizable messages from the participants to the head participant. Another factor to consider in using this approach is that the malfunctioning of

the head participant process causes the loss of the entire CAT function. Various ways to avoid such a loss of the CAT function are conceivable but they all represent additional costs.

3.2 Decentralized CAT

In this approach each participant performs its own acceptance test, and the participants exchange their results with each other. Therefore, every process receives the results of other participants and figures out by itself the result of the "non-local" acceptance test.

One of the advantages of this approach is that all participants have the symmetric structure. This makes it simple to implement. However, decomposition of the CAT function is sometimes a costly burden on the programmer. Although automated decomposition is conceivable, its practicality requires further study. Also, all N messages needed for CAT are broadcast messages, i.e., each message has N destination processes including the sender itself. Therefore, if an efficient broadcasting channel is not available, the number of CAT-related messages exchanged among participants may become very large, although each message will be short. (That is, the messages include "pass" or "fail" information only.) Therefore, in such an environment the communication cost becomes very high.

3.3 Semi-centralized CAT

This approach compromises the above two approaches in such a way that the "local" acceptance test is done by each participant and the "non-local" CAT result is determined by the head participant. That is, each participant performs its own acceptance test and sends the result to the head participant. The head participant judges the success or failure of the CAT depending upon whether all the reports received are success reports or not, and then broadcasts the CAT result. In order to facilitate speedy recovery this approach should be implemented whenever possible in such a way that a failure report of a participant is made in a broadcast form so that all participants may begin their rollback actions immediately. The main advantage of this approach is the small communication overhead; all N messages are short messages and among them N-1 messages are one-to-one messages. The semi-centralized approach, however, shares one deficiency with the decentralized CAT approach. That is, it is necessary to decompose the CAT function.

3.4 Discussion

Table 1 summarizes the number of messages needed for CAT under each approach. As discussed in the previous sections the total number of messages is the same for all three cases, but the messages communicated under the three approaches are different in length and number of des-

tination processes. For example, as mentioned earlier, the messages from the participants to the head participant under the centralized CAT approach are of the sizable type because the messages include the values of the variables needed for CAT. On the other hand, the CAT-related messages required in the semi-centralized and decentralized CAT approaches include "pass" or "fail" information only. Therefore, the size of each message is very short and fixed. Nevertheless, all the messages in the decentralized CAT approach need to be broadcast whereas in the centralized and semi-centralized CAT approaches only one message (the non-local CAT result message broadcast by the head participant) needs to be. It seems reasonable to conclude that the semi-centralized CAT approach is the best in terms of message traffic.

Message type \ Policy	Centralized	Decentralized	Semi-centralized
Total number of messages for CAT	(N - 1)(s) + 1(sn)	N(sn)	(N - 1)(l) + 1(sn)
one-to-one large* messages (l)	N - 1		
one-to-one small messages (s)			N - 1
one-to-n small messages (sn)	1	N	1
Maximum number of CAT-related messages from/to a participant	2	N	2
Head participant case	N		N

Number of participant = N
l: long message
s: short message
sn: short, broadcast message

* This message contains the values of the variables needed for CAT.

Table 1. Comparison in number of messages for CAT.

In contrast to the decentralized or semi-centralized CAT approach, the centralized CAT approach does not require any local acceptance test to be done by each participant. This possibly degrades detection and recovery performance because the CAT is not evaluated until all the

participants complete the conversation tasks. Under the decentralized or semi-centralized CAT approach, it is possible to have participants abandon the conversation tasks if one of the participants has failed in its local acceptance test. This reduces unnecessary computation time although the amount of gain is highly application-dependent. The relatively high fault latency associated with the centralized CAT approach can be a serious drawback in many safety-critical applications such as that to be discussed in Section 6.

The choice should also be made based on the following two factors: (1) characteristics of the application program such as process structure, reliability consideration, etc., and (2) characteristics of the communication system such as network topology, communication medium, and protocol used. Also, the conversation structuring scheme used by the program designer favors a certain CAT execution approach. This will be further elaborated in Sections 4 and 5.

4. Implementation of the NLRB Scheme in Message-based DCS's

This and the next sections describe how some structuring schemes used by the program designer and the execution approaches discussed in preceding sections can be used in combinations in implementing conversations in message-based DCS's. The structuring schemes selected for discussion in this paper are the Name-Linked Recovery Block (NLRB) scheme and the Abstract Data Type (ADT-) Conversation scheme described in [Kim82,Rus79].

Message-based DCS's considered in these two sections have the following characteristics:
(1) Each node contains a processor, a local memory, and an IO handler.
(2) Each node runs one or more software processes.
(3) Communication between processes in different nodes is done through a bus with the aid of IO handlers. (Broadcasting is facilitated).
(4) The IO handler in the destination node receives and puts a message into the mailbox of the destination process.

4.1 NLRB scheme

The basic idea of the NLRB scheme is to extend the recovery block (RB) construct with a conversation identifier field as follows [Kim82,Rus79]:

```
[ conv  C:]
ensure      T
by          B1
else by     B2
...
else by     Bn
else error
```

where C is the conversation identifier, T the acceptance test, and Bi, $1 \le i \le n$, the try blocks. The set of name-linked RB's, each executed by a different process but having the same conversation identifier, compose a conversation construct.

Although this scheme has several deficiencies as pointed out in [Kim82], the scheme is believed to be suitable in some distributed environments for the following reason. In some message-based DCS's, nodes are geographically dispersed. It is very likely that in such application environments processes on different nodes are designed largely independently. In such an environment the NLRB scheme is more natural for use than other schemes such as the ADT-conversation scheme [Kim82].

4.2 Syntax adopted and message format

The following syntax is an extension of the original NLRB syntax and the extended part is the "participant" field.

```
[ conv C:]
participants  PROCA, PROCB, ...
ensure        T
by            B1
else by       B2
...
else by       Bn
else error
```

where PROCA, PROCB, ... are the process ids of the conversation participants. (Each process has a unique process id in the system.)

In the following subsections, execution of NLRB's under the two different exit control approaches and the two different CAT execution approaches, semi-centralized and decentralized, are considered. The centralized CAT approach was not considered because each participant of the NLRB conversation is designed to have its own acceptance test routine and placing all the participants' acceptance test routines in one node did not seem competitive with other two approaches in terms of resulting execution performance.

The generic message format adopted is shown in Figure 9. Note that it is of a multicast type sent to more than one destination process. By doing this the number of messages communicated among the processes can be reduced. Messages are classified largely into two types: normal message and CAT result message. In the case of a CAT result message, the data field is empty. As will be shown in the following subsections, different types of CAT result messages are required for different implementations.

4.3 Approach N1 for execution of NLRB's: Synchronous exit and semi-centralized CAT execution

Under the synchronous exit approach, history logging is not necessary because a participant can exit the conversation only when all the participants pass their acceptance tests. Therefore, its implementation is relatively simple.

The types of CAT result messages used in this execution approach are as follows:
(1) success notice (SU): This message is sent to the head participant when the participant has passed its local acceptance test.

ML:	message length in number of bytes	(2 bytes)
T:	message type	(1 byte)
N:	number of destination process	(1 byte)
D:	destination process id's	(1 bytes per each destination)
S:	source process id	(1 byte)
Data:	contents	(m bytes)
EOM:	end of message	(2 bytes)

Figure 9. Message format.

(2) failure notice (FA): This message is sent to the head participant when the participant has failed in its local acceptance test.
(3) non-local success notice (GS): This message is broadcast when the head participant has concluded the CAT to be a success.
(4) non-local failure notice (GF): This message is broadcast when the head participant has concluded the CAT to be a failure.

In normal cases (i.e., when there is no fault) N messages are needed for each conversation where the number of the participants is N; those are N-1 SU messages from the participants to the head participant and one GS message from the head participant to others. For example, if there are six participants in a conversation, six messages are needed for each conversation: five one-to-one messages are 8 bytes-long each and one broadcast message is 12 bytes-long (because this broadcast message has five destinations). If the head participant receives an FA message, it immediately notifies other participants by broadcasting the GF message and then all participants should roll back. Therefore, in order to minimize the recovery time it is desirable to incorporate an interrupt mechanism for receiving an FA or GF message.

4.4 Approach N2 for execution of NLRB's: Synchronous exit and decentralized CAT execution

The execution approach is almost the same as the previous one (Approach N1). As mentioned earlier, however, there is no head participant with the decentralized CAT approach. The success or failure notice from each participant is broadcast to all other participant processes. Therefore, in normal cases N SU messages are needed for each conversation. For example, with six participants six broadcast messages are needed and each message is 12 bytes-long. Upon receiving an FA message the participant rolls back for retry. Besides the SU and FA messages, no other types of CAT result messages are required.

4.5 Approach N3 for execution of NLRB's: Asynchronous exit and semi-centralized CAT execution

As discussed in Section 2, implementation of asynchronously exited conversations is much more costly. Also, many more types of CAT result messages are needed than in the case of the synchronous exit:

(1) complete local validation notice (CV): This message is sent to the head participant when a process that is in the irrevocable state has passed its acceptance test.

(2) partial local validation notice (PV): This message is sent to the head participant when a revocable process has passed its acceptance test.

(3) failure notice (FA): This message is sent to the head participant when a process has failed in its acceptance test.

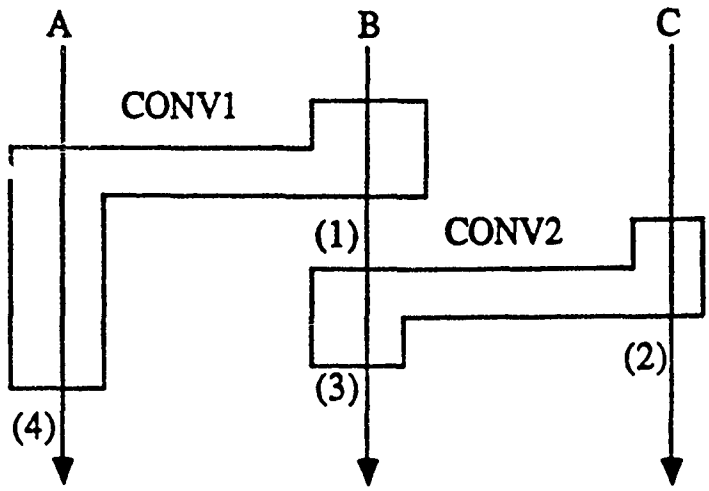
(4) non-local success notice (GS): This message is broadcast when the head participant has concluded the CAT to be a success. That is, the conversation is completely validated.

(5) non-local failure notice (GF): This message is broadcast when the head participant has concluded the CAT to be a failure.

(6) rollback notice (RO): This message is sent to/from the head participant of a conversation when a participant process learns that the conversation which it participated in earlier has become a failure.

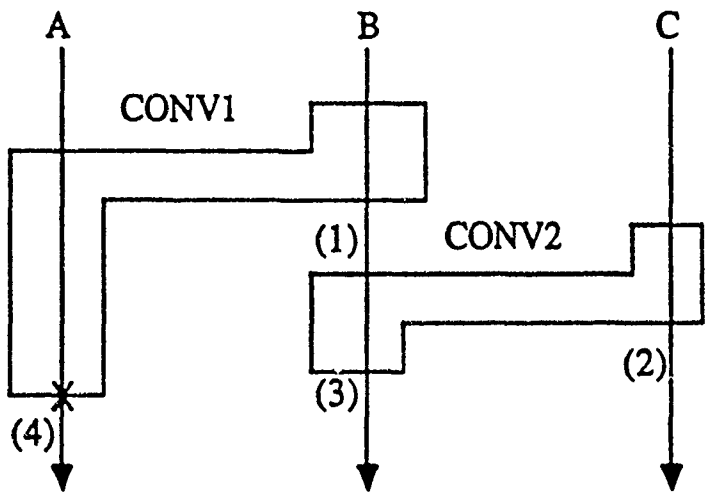
(Note: The PV message is not an absolute necessity. However, it is believed that the PV message is helpful in implementing and testing the system.)

Figure 10 shows two different execution scenarios under N3. Process A is the head participant of CONV1 and Process C is the head participant of CONV2. In both scenarios Processes B and C complete CONV2 with no failure before Process A completes CONV1. Therefore, CONV2 is not completely validated until CONV1 is since CONV1 is logically before CONV2. In the first scenario (shown in Figure 10.a), by the time Process A passes CAT of CONV1 both CONV1 and CONV2 become completely validated. Consequently, in Process B the conversation records for CONV1 and CONV2 are removed. In the second scenario (shown in Figure 10.b), on the other hand, Process A fails at (4), which results in CONV2 being nullified since Process B has to roll back to the recovery line of CONV1. Process B sends the RO message to Process C (which is the head participant of CONV2) and retries CONV1 with the next alternate try block. Later, Processes B and C are allowed to use their primary try blocks when they reenter CONV2.



History	Message	Conversation State CONV1	CONV2
(1) B passes CAT of CONV1	B-->A: CV(CONV1)	og	--
(2) C passes CAT of CONV2		og	og
(3) B passes CAT of CONV2	B-->C: PV(CONV2)	og	pv
(4) A passes CAT of CONV1	A-->B: GS(CONV1)	cv	pv
	B-->C: CV(CONV2)	cv	cv
	C-->B: GS(CONV2)	cv	cv

Figure 10.a. Asynchronously exited conversation: Scenario I.



History	Message	Conversation State CONV1	CONV2
(1) B passes CAT of CONV1	B-->A: CV(CONV1)	og	--
(2) C passes CAT of CONV2		og	og
(3) B passes CAT of CONV2	B-->C: PV(CONV2)	og	pv
(4) A fails CAT of CONV1	A-->B: FA(CONV1)	sf	pv
	B-->C: RO(CONV2)	sf	nu
	C-->B: RO(CONV2)	sf	nu

Figure 10.b. Asynchronously exited conversation: Scenario II.

By allowing asynchronous exit the number of messages needed for each conversation may increase. That is, in normal cases the participants exchange N-1 CV messages, one GS message, and up to N-1 PV messages. For example, if there are six participants, five CV messages are 8 bytes-long each, one GT message is 12 bytes-long, and each PV message is 8 bytes-long. The number of PV messages needed depends on the number of revocable processes in the participants of the conversation. This extra overhead may become serious as the lookahead levels increase.

4.6 Approach N4 for execution of NLRB's: Asynchronous exit and decentralized CAT execution

This execution approach is almost the same as the previous case. Since there is no head participant under the decentralized CAT approach, the result of the non-local acceptance test is not broadcast. Consequently, GS and GF messages are not required. In normal cases N CV messages (and up to N PV messages) are broadcast. Therefore, with six participants all six CV messages are 12 bytes-long. Each PV or RO message is also 12 bytes-long.

4.7 Discussion

As expected, asynchronously exited conversations require more messages than synchronously exited conversations, although the overhead does not seem serious with one or two-conversation lookahead. The message traffic incurred under the semi-centralized CAT approach and that under the decentralized CAT approach are almost the same. However, the load on each participant (due to incoming and outgoing messages) in the decentralized CAT approach is higher than that in the semi-centralized CAT approach as shown in Table 1. The difference becomes larger as the number of participants increases.

We did not examine the centralized CAT execution cases closely in this section since the NLRB scheme is most likely applicable to "loosely coupled" network environments where processes on different nodes are designed largely independently. Consequently, the CAT execution by one process/node is not natural in such environments. Moreover, the centralized CAT approach requires long messages which may result in too much communication overhead in loosely coupled network environments. Nevertheless, it is possible to apply the implementation strategies of the centralized CAT approach which will be discussed for the case of ADT-Conversation in Section 5 to the NLRB scheme.

5. Implementation of the ADT-Conversation Scheme in Message-Based DCS's

5.1 ADT-Conversation scheme

The Abstract Data Type (ADT-) Conversation scheme was proposed to remedy the shortcomings of the NLRB scheme [Kim82], which stems from the scattered appearance of the constituent RB's of a conversation structure in the program text. In the ADT-Conversation scheme, the conversation construct is structured in the form of an abstract data type.

A possible syntactic structure is shown in Figure 11. The participant enters a conversation by calling a procedure, say CONV.PROCA, of which the name and the formal parameters are listed in the participant area. The role of CO (conversation object) in the figure is to facilitate interprocess communication within the associated conversation. In other words, CO is accessible only within the conversation. This prevents information smuggling by a process participating in the conversation.

Basically, there are two options in executing the CAT. One is to decompose the CAT into segments, each executed by a different participating process. The other is to have one process execute the entire CAT after all the participating processes have completed their executions of conversation tasks. Once the code for the CAT is decomposed (i.e., the first option is adopted), the non-local acceptance test is done with either the decentralized or semi-centralized approach. The implementation strategies for these cases should be the same as those discussed in the previous section (i.e., NLRB scheme cases). Therefore, in the following subsections 5.3 and 5.4 strategies for implementing the ADT-Conversation scheme with centralized CAT approach in combination with either the synchronous or the asynchronous exit approach are discussed.

5.2 Syntax adopted and message format

The syntax given in Figure 11 can be incorporated into most of the target implementation languages, although some variations are possible. The ADT-Conversation incorporated into Path Pascal [Kol80] was implemented by the authors [Kim85]. The same message format given in the previous section (Figure 9) is used here again.

```
type C = conversation
   <const & type declaration>              "can declare nested conversation"
   participants
         PROCA ( "formal parameters" );
         PROCB (  ..................................  );
         ..........
   var
         CO: c-object-type;                "conversation object declaration"
         ..........
   <CAT function declaration>              "conversation acceptance test"
   <procedures & functions declaration>
   ensure CAT
   by begin                                "primary interacting session"
         PROCA: begin ..... end;
         PROCB: begin ..... end;
         ..........
      end;
   elseby begin                            "alternate interacting ..
         PROCA: begin ..... end;
         PROCB: begin ..... end;
         ..............
      end;
      ..............
   elseerror
endconversation;
```

Figure 11. ADT Conversation.

5.3 Approach A1 for execution of ADT-Conversations: Synchronous exit and centralized CAT execution

In the centralized CAT approach, only one participant executes the acceptance test and broadcasts the result. Therefore, only two types of CAT result messages are needed since no local acceptance test is done by the participant.

(1) non-local success notice (GS): This message is broadcast when the CAT has succeeded.

(2) non-local failure notice (GF): This message is broadcast when the CAT has failed.

However, as discussed in Section 3, the participants send the values of the variables needed for CAT to the head participant. Since these messages are longer than simple "pass" or "fail" messages the communication overhead of this approach is higher than other approaches. For example, suppose that there are six participants, including the head participant which executes CAT, and each participant provides 50 bytes data for CAT execution. Then each conversation requires five 58 bytes-long messages and one 12 bytes-long message. Moreover, in general, recovery time under this approach is longer than under the others because CAT cannot be performed until all participants complete the conversation tasks and provide information needed for CAT. (Under the decentralized or semi-centralized CAT approach it is possible to have the participants abandon their conversation tasks if one participant has failed in its local acceptance test.)

5.4 Approach A2 for execution of ADT-Conversations: Asynchronous exit and centralized CAT execution

With asynchronous exit even the pass of the CAT does not make the conversation completely validated unless all the participants are irrevocable. Therefore, the non-local success notice is deferred if there is at least one revocable participant. Three types of CAT result messages are needed.

(1) non-local success notice (GS): This message is broadcast when the CAT has succeeded and all the participants are irrevocable.

(2) non-local failure notice (GF): This message is broadcast when the CAT has failed.

(3) rollback notice (RO): This message is sent to the head participant of a conversation (say X) when a participant process learns that the conversation (say Y) which it participated earlier has become a failure.

The messages needed for CAT in this approach are basically the same as those in the previous approach (Approach A1). That is, in normal case five 58 bytes-long messages and one 12 bytes-long message are needed assuming that there are six participants and each participant provides 50 bytes data for CAT execution.

5.5 Discussion

The ADT-Conversation scheme has a number of advantages over the NLRB scheme. Among others, (1) each interacting session is presented as a single unit in the program text and thus easier to read and (2) it is not necessary to decompose the CAT into distributed routines of which collective effect is generally harder to comprehend. Also, all six implementation approaches

(combinations of three different CAT approaches and both synchronous and asynchronous exit cases) are applicable. However, it seems natural to have the centralized CAT for the ADT-Conversation scheme because decomposition of CAT could be a costly burden on the program designer. Table 2 summarizes how three CAT execution approaches are applicable to the NLRB scheme and the ADT-Conversation scheme.

Schemes \ CAT	Centralized CAT	Decentralized CAT	Semi-centralized CAT
NLNB	less applicable	most applicable	applicable
ADT-Conversation	most applicable	applicable	applicable

Table 2. Applicability of three CAT approaches.

Approach \ Overhead	Type of messages	Time overhead (10 µsec fixed time)	Time overhead (100 µsec fixed time)
Centralized CAT			
Synchronous Exit	5(1-to-1) + 1(broadcast)	2476 µsec	3016 µsec
Asynchronous Exit	5(1-to-1) + 1(broadcast)	2476 µsec	3016 µsec
Decentralized CAT			
Synchronous Exit	6(broadcast)	636 µsec	1176 µsec
Asynchronous Exit	9(broadcast)	954 µsec	1764 µsec
Semi-entralized CAT			
Synchronous Exit	5(1-to-1) + 1(broadcast)	476 µsec	1016 µsec
Asynchronous Exit	8(1-to-1) + 1(broadcast)	698 µsec	1508 µsec

* 3 PV messages are assumed in the asynchronous exit case.

Table 3. Communication overhead of six implementation approaches.

Table 3 summarizes the communication overhead of six implementation approaches (combination of three different CAT approaches and both synchronous and asynchronous exit cases). We assume here that there are six participants and each participant provides 50 bytes data to the head participant in the case of the centralized CAT approach. We also assume that transmission speed is 1 μsec per bit (i.e., 1 Mbps transmission). This table shows communication time overhead of two different cases: in one case the fixed time overhead for each communication is 10 μsec and in the second case this overhead is 100 μsec. As shown in the table the time overhead under the centralized CAT approach is almost three times of that of the decentralized or semi-centralized CAT approach even with the 100 μsec fixed time overhead. (As this fixed time overhead decreases, the ratio will increase.) It also shows that the decentralized CAT approach requires a little more overhead than the semi-centralized CAT approach.

6. Simplified Unmanned Vehicle System: An Example

The previous sections dealt with general strategies for implementing the conversation scheme into message-based DCS's. In this section, a simplified unmanned vehicle system (SUVS) is used to illustrate the conversation implementation strategies.

6.1 Scenario

The SUVS consists of three different sets of tasks, i.e., sensor tasks (or sensors), analyzer tasks (or analyzers), and actuator tasks (or actuators).
(1) Sensors are input devices such as speed meter, engine thermometer, direction indicator, vision sensor, and road surface sensor. They periodically provide data to the analyzer tasks.
(2) Analyzers process sensor data and include speed analyzer, direction analyzer, vision analyzer, and surface analyzer. They make decisions which are forwarded to the actuators. They also exchange information among themselves.
(3) Actuators are output devices such as brake, accelerator, handle, and camera handlers. They receive commands from the analyzers and cause the controlled objects to change their states.

The information flow among these tasks is shown in Figure 12. In the following subsection we illustrate the strategies for incorporation of the conversation scheme into the system. Our discussion focuses on the implementation of the analyzer tasks.

6.2 Implementation strategies

6.2.1 System architecture

A bus-structured computer network, as shown in Figure 13, is assumed for this implementation. Each node has its own local memory and database which runs a single analyzer task. Tasks communicate with each other via message passing. Nodes are also connected to sensors and actuators.

6.2.2 Incorporation of the conversation scheme

In order to incorporate conversations into a real system, the characteristics of the system (in terms of interaction among tasks) should be carefully analyzed. This SUVS has the following characteristics. First, the decision made by one analyzer affects the decision of other analyzer(s). For example, before the direction of a car is changed, the speed may have to be reduced, if the current speed is too fast. This means that the direction analyzer has to cooperate with the speed

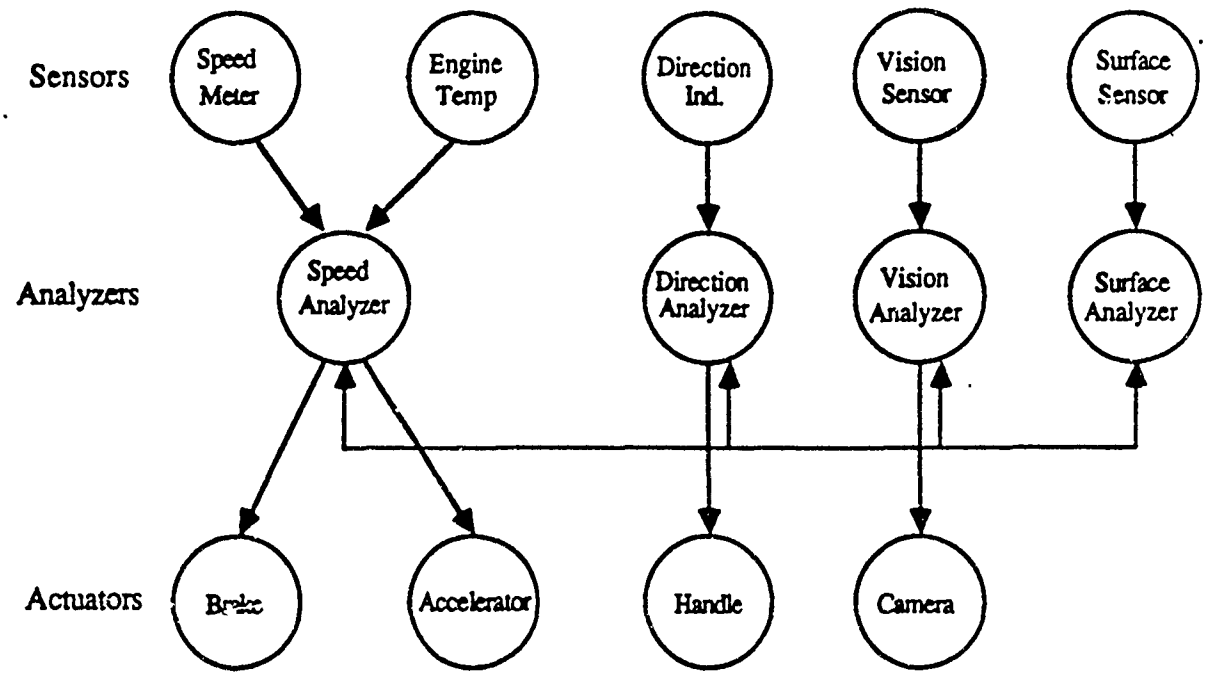


Figure 12. Information flow of Simplified Unmanned Vehicle System (SUVS).

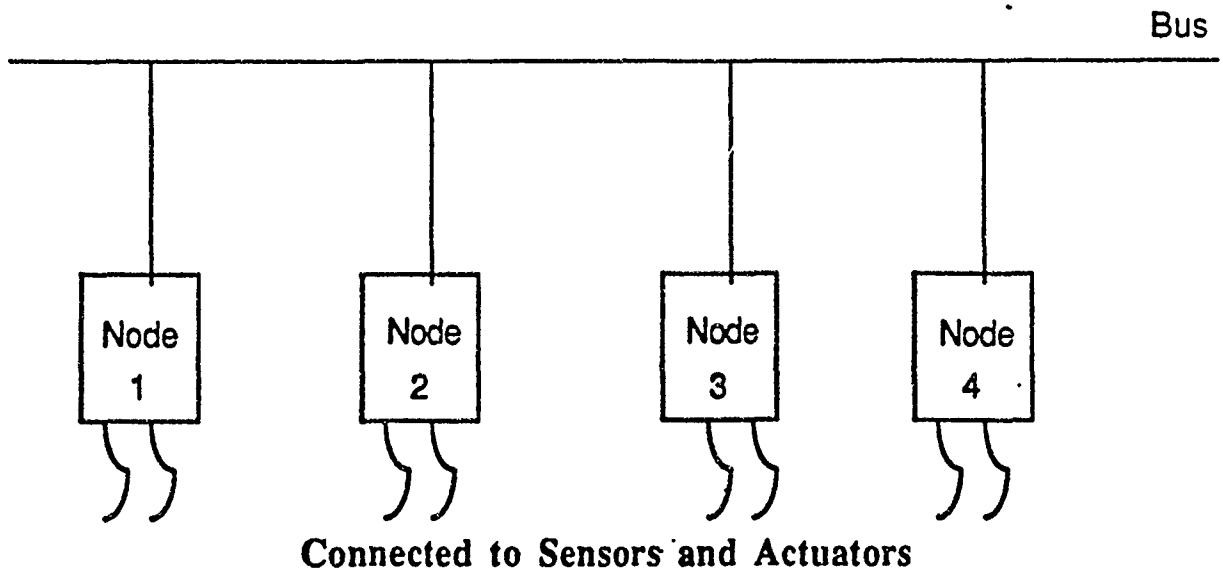


Figure 13. Computer network to implement the SUVS.

analyzer. Second, the actions taken by the actuators are not revocable. There may be cases where we cannot compensate or change, even if we find a mistake immediately after an action was done. Therefore, the output actions should be very carefully taken. Finally, any decision should be made within a specified time under all circumstances. (Otherwise, the effect is the same as driving a car while sleeping.)

The above three characteristics requires cooperation among the analyzer tasks to properly control the real-time response of the system. Therefore, the following conversation structure seems useful.

(1) Four analyzer tasks, i.e., speed, direction, vision, and surface analyzers, cooperate (i.e., exchange information and preliminary decisions) to make decisions on any action.

(2) Output (to actuators) is made only when all of the analyzer tasks agree that the system is in a safe state.

(3) Upon disagreement, try again with the alternate algorithms provided.

(4) If they don't reach any final agreement within a specified time or they have failed in all alternate algorithms, the system goes into an emergency mode and tries to stop the car in the safest and fastest way.

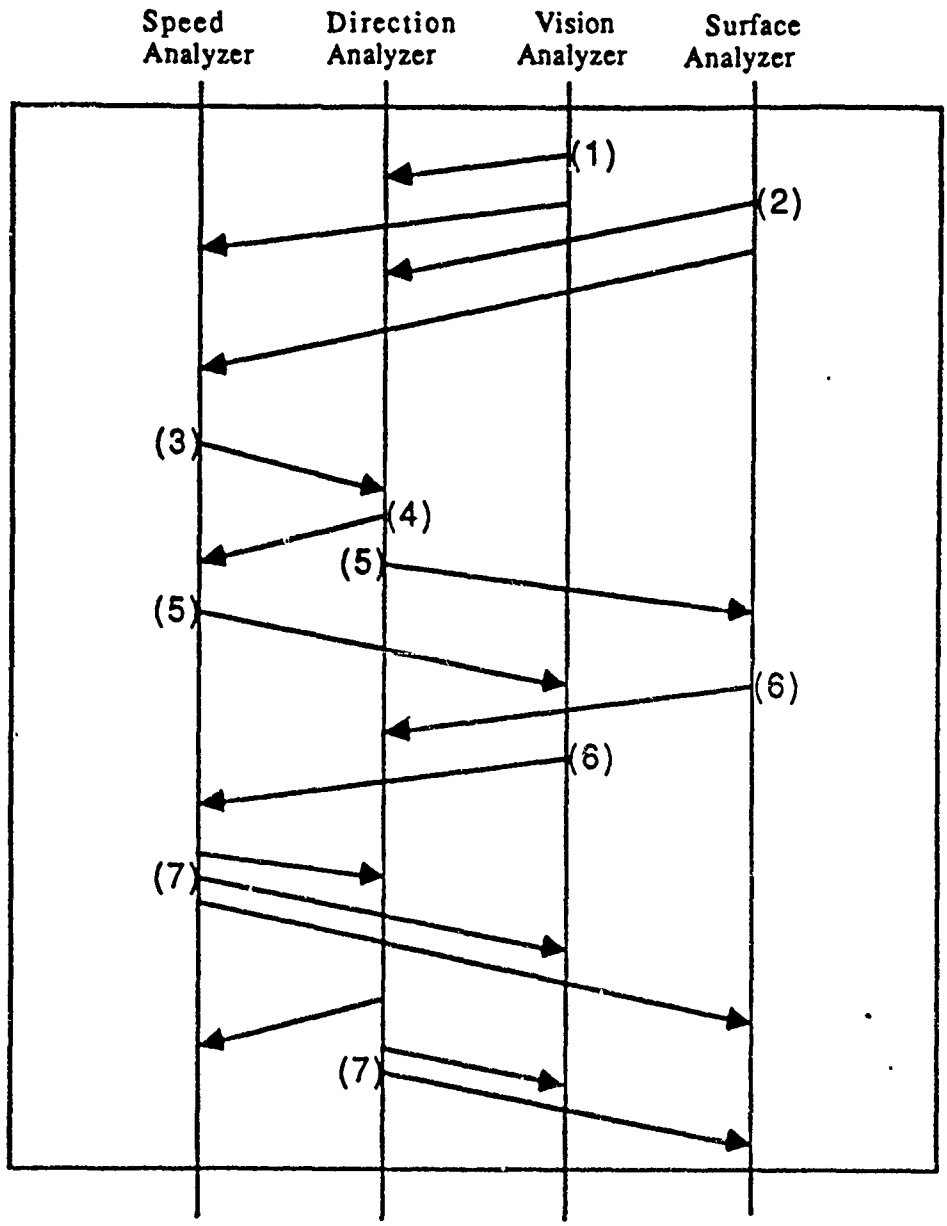
The conceptual conversation structure and possible information exchanged among tasks are shown in Figure 14. Note that the preliminary decisions made by the tasks are broadcast so that this information may be used for the conversation acceptance test. The acceptability criteria may include

(1) whether the decisions made by the speed analyzer and the direction analyzer conflict with each other,

(2) whether the decision made by the vision analyzer conflicts with the request made by the speed and/or direction analyzer, and

(3) whether the analyzers have made decisions based on correct information.

Each analyzer also needs to check whether the current inputs are consistent with the outputs made before. For example, if the output action is "reducing speed", then we expect a reduced speed after a while. If the inputs are not consistent, we should suspect either actuator or sensor, or both. Therefore, an emergency action is required. Further details on the semantics of this conversation are given later in section 6.2.5.



(1) Vision information
(2) Surface information
(3) Acceleration information
(4) Momentum information
(5) Request for more information needed
(6) Response on request
(7) preliminary decisions

Figure 14. Conceptual conversation structure and information exchange among the tasks.

6.2.3 Exit Control

Intuitively, the synchronously exited conversation scheme is suitable for the conversation sketched in the previous section because the conversation is followed by critical output actions of the analyzers. The outputs (to the actuators) made by the analyzers are irrevocable and sometimes critical, thus making it dangerous to allow asynchronous exit in most cases.

However, we may need to allow a special kind of asynchronous exit (i.e., to send the output to the actuator before the non-local acceptance test is done) to handle an emergency situation. For example the speed analyzer has decided to reduce the speed quickly after it received the information about an unexpected object from the vision analyzer and/or surface analyzer. In such a case, all the analyzers abandon their current execution and restart the conversation based on the new information.

6.2.4 CAT execution

Although all three CAT execution approaches (centralized, decentralized, and semi-centralized) are applicable, the decentralized and semi-centralized CAT approaches seem more suitable mainly because safety is the major concern in this type of applications and fast recovery is very important. In other word, the relatively high fault latency characteristics makes the centralized CAT approach less attractive in this application. Also, since "fail-stop", i.e., to stop the car if the system does not reach any final agreement, is allowed as a safe emergency action in SUVS, it is beneficial to utilize a conservative approach which may generate more "false alarms" than other approaches. False alarms do little harms but late alarms or absence of necessary alarms can be catastrophic. The decentralized and semi-cetralized CAT approaches are more conservative approaches in the sense that under the approaches the system does not suffer from a possible catastrophe due to abnormal behavior of the sole generator of alarms in the centralized CAT approach, i.e., the head participant. Upon passing their acceptance tests the analyzers output proper messages (based on the decisions made) to the actuators. Otherwise, all analyzers roll back and try again with alternate try blocks. In this kind of cases there is generally no need to use the previous input data (which were already used for the execution) if the new input data are available.

6.2.5 The fault-tolerant SUVS

In this subsection we describe the fault-tolerant SUVS (i.e., SUVS extended with conversations) in more detail. Figure 15 depicts the high-level logic of the analyzer task. (This logic is applicable to all four analyzer tasks.) The analyzer receives new input data periodically (every 10 msec in this illustration) from the corresponding sensor(s). The input data are checked

to see whether they are consistent with the outputs made before. Then the data are exchanged among the analyzers and used to determine the next outputs.

Once the decision is made by the analyzer it is broadcast to other analyzers. These preliminary decision results are used in parts of the CAT performed under the decentralized (or semi-cetralized) execution approach. The exchanged information is used to ensure that the decisions made by the analyzers should not conflict with each other. Then the CAT result (performed by each analyzer) is broadcast to facilitate the non-local acceptance test. If all have passed their acceptance tests the preliminary decisions made are forwarded to the actuators. Otherwise, the analyzers roll back and retry with the alternate interacting session. For every cycle of the execution the timer is set. If a timeout occurs, an "emergency routine" is invoked. Once it is invoked, normal operation is suspended and an attempt is made to stop the car in the safest and fastest way.

```
task analyzer;

    every 10 msec do  ( on "timeout" => invoke emergency routine; )
        receive input data from the sensor(s) and analyze them;
                                          .      -- for validity and consistency check
        exchange the data among the analyzers;
        make a decision;
        exchange the decisions made along with the input data used among the analyzers;
        perform CAT;
        if "pass" => forward the decision to the actuator(s);
        if "fail" => roll back and retry;
    end do every;

end task analyzer;
```

Figure 15. High-level logic of the analyzer task in SUVS.

In principle, the alternate interacting session (i.e., alternate try blocks of the analyzers) should be designed such that it may produce acceptable results for the cases where the primary interacting session (i.e., primary try blocks of the analyzers) fails to do so. Although designing efficient alternate interacting sessions is outside the scope of this paper (and also it is somewhat application-dependent) we briefly sketch a systematic approach which, we believe, is applicable to this type of applications. The major difference between the primary try blocks and the alternate try blocks in SUVS should be in the way the decisions are made. And those decisions are made based on several factors. For example, the speed analyzer makes a decision based on current speed, RPM (revolutions per minute) of the engine, road condition (e.g., wet or dry), curve of the road, and existence of objects in front and if exists, characteristics (e.g., moving speeds) of the objects.

Therefore, one way to design alternate try blocks is to apply the decision factors in different sequences.

Figure 16 shows two different approaches to designing the speed analyzer tasks. In Figure 16.a the road condition is first examined; then the curve status of the road is examined and so on. On the other hand, in Figure 16.b, the current speed is first examined; then the existence of an object is examined and so on. By doing so we can systematically produce multiple versions of software. Such produced versions are still considerably diverse in the detailed logics used and also have substantially different chances of encountering overflow/underflow conditions due to the diversity in the sequence of calculations used.

```
task primary speed analyzer;

....
-- decision making routine of the primary try block

    if road condition is dry
        => if the road is straight
            => if ...
                ....

    else if road condition is wet of degree 1
        => if ...
```

Figure 16.a. The primary try block of the speed analyzer in SUVS.

```
task alternate speed analyzer;

....
-- decision making routine of the alternate try block

    if 0 < current speed <= 5 mph
        => if there is no object in front
            => if ...
                ....

    else if 5 < current speed <= 10
        => if ...
```

Figure 16.b. The alternate try block of the speed analyzer in SUVS.

A major portion of the acceptance test in the SUVS is to check whether (1) the decisions are made based on correct information, (2) the decision made locally is reasonable with respect to both the recently observed condition of the car and the laws of physics, and (3) the decisions do not conflict with each other. The first part of the test (which is trivial in nature in comparison to the

other two parts) can be facilitated by sending the input data (which were used to make decisions) along with the preliminary decisions made. For example, the input data from the speed meter is initially received by the speed analyzer for every cycle of the execution. This data is checked and then broadcast to other analyzers. The data may then be used by other analyzers in reaching certain preliminary decisions. Therefore, in the first part of the CAT the speed analyzer checks whether the speed data received from other analyzers together with their preliminary decisions are exactly the same as the original data that it received from the speed meter and has kept since. By doing so we can detect possible faults due to communication failure and/or memory failure. The second and more important part of the CAT is largely to check if the preliminary local decision falls within a reasonable range. For example, if the preliminary decision on the acceleration to be made is beyond the capacity of the car, clearly a computation error can be suspected. Actually, this reasonableness test of the local preliminary decision can be performed even before it is sent to other analyzers. The third part (i.e., checking the possibility of conflict) can be viewed as a "non-local" logic test. That is, the decisions made by the analyzers are examined to see if there is any conflict among them. For example, as shown in Figure 17, decisions made by the speed analyzer and the direction analyzer should not conflict with each other.

```
task analyzer;

    ....
    -- conversation acceptance test (CAT)

        if (attached data which were used to make decisions = original data)
            => if acceleration* = +2 and direction** = +3
                => if (current speed < 20 mph) and (current RPM < 1000)
                    => if ....
                        => CAT is "pass"
                    => else if ...
                        => CAT is "fail"
        ....

        exchange the result of local acceptance test;
        perform the non-local acceptance test;
```

* "acceleration", the output of the speed analyzer, is an integer value which indicates the acceleration ranged between -3 and +3. (Zero means no change.)

** "direction", the output of the direction analyzer, is an integer value which indicates the angle ranged between -9 and +9. (Zero means no change.)

Figure 17. Conversation acceptance test in SUVS.

6.2.6 Run-time support

One of the important run-time functions is to facilitate efficient and reliable communication among tasks. Since tasks run under tight synchronization, especially for CAT-related messages, message delay blocks the execution of other task(s). This may lead to the degradation of the system performance significantly. Furthermore, most messages are time-sensitive, i.e., the validity of a message depends on time. Therefore, protocols should be designed in such a way to support reliable and real-time communication. (For real-time communication the timing behavior of the protocol should be predictable.)

The system should also handle timeouts associated with conversations as well as those with protocols. The decision made by the analyzer becomes obsolete after a certain time period. Hence, absence of a report (local acceptance test result) from a participant within a timeout period during the CAT execution phase should be treated as the failure of the CAT. Then, all tasks should roll back and retry with their alternate try blocks. Finally, as mentioned in Section 4.7, an interrupt mechanism is required for fast rollback of tasks, i.e., for minimizing the waste of time.

6.3 Discussion

In this section, an unmanned vehicle system was used to illustrate the factors to be considered in incorporating the conversation scheme into real-time control systems. The centralized CAT approach is not suitable for this application because of its relatively high fault latency and high communication overhead characteristics. This means that if the ADT-Conversation structuring approach were to be used, decomposition of the CAT for decentralized or semi-cetralized execution must be performed either by the program designer or by means of an automated tool. Also, since the entire control cycle is captured in one conversation, the synchronous exit approach is natural in this application. If the control cycle was implemented in the form of a series of conversations, then it might be possible to exploit the asynchronous exit approach in executing the conversations except the last one which is followed by actuator output actions. The approaches to implementation of the conversation scheme outlined in this section are believed to be applicable to many safety-critical applications.

7. Summary

This paper presented several different approaches for implementing the conversation scheme in message-based DCS's. Important implementation factors such as the control of exits of processes upon completion of their conversation tasks and the approach to execution of the conversation acceptance test were considered. A new efficient approach to run-time management of recovery information based on an extension of the recovery cache scheme was also proposed. Both exit control strategies, synchronous and asynchronous exits, and three different approaches to execution of CAT, centralized, decentralized, and semi-centralized, have been examined and compared in terms of system performance and implementation cost. Since each approach has merits and deficiencies, it is hard to say that one approach is simply better than another. Moreover, the implementation strategy should be carefully chosen based on the characteristics of the application system such as network topology, communication cost, etc.

However, it seems that the asynchronous exit approach is generally better than the synchronous exit approach. The former provides a higher performance during error-free execution than the latter. If the NLRB structuring approach is used, then the semi-centralized CAT approach or the decentralized CAT approach are more attractive than the centralized approach. On the other hand, if the ADT-conversation structuring approach is used, then the selection of a good strategy for execution of a CAT depends on whether one can afford the effort required to decompose the CAT into distributed acceptance tests. If so, the semi-centralized or decentralized approach is more attractive and otherwise, the centralized CAT execution is the only choice.

Incorporation of the timeout capability into the conversation scheme is another area studied [Hec76]. The crash of a participant (or a node) can result in the lockup of several other nodes if the timeout mechanism is not used. Since each participant enters the conversation asynchronously, the timeout period is an important design parameter, and an effective technique for determination of a proper timeout period needs to be developed. Integration of the conversation scheme and other established fault tolerance schemes [Bha87,Kim84,Toy87] is also an important area for future research.

Through analytic modeling studies, we have obtained some understanding of system performance behavior under various workload conditions. However, in order to obtain "real" data on implementation cost and system performance, experimental work and field experiences are necessary. Testbed-based evaluation [Chu87] of the proposed approaches in the context of a real world application, such as the unmanned vehicle system illustrated in this paper, is regarded as a highly worthwhile research topic. Such efforts will also provide further insights into the types of

applications for which the conversation scheme become a cost-effective approach to reliability enhancement.

8. References

[And76] Anderson, T. and Kerr, R., "Recovery Blocks in Action: A System Supporting High Reliability", *Proc. 2nd Int'l Conf. on Software Engineering*, 1976, pp. 447-457.

[Bha87] Bhargava, B., editor, *'Concurrency and Reliability in Distributed Systems'*, Van Nostrand and Reinhold, 1987.

[Chu87] Chu, W.W., et al., "Testbed-Based Evaluation of Design Techniques for Fault-Tolerant Real-Time Distributed Computer Systems", *Proc. of the IEEE*, Vol. 75, No. 5, May 1987, pp.649-667.

[Cam83] Campbell, R.H., Anderson, T., and Randell, B., "Practical Fault Tolerant Software for Asynchronous Systems", *Proc. SAFECOM 83*, Cambridge, Oct. 1983, pp.59-65.

[Gre85] Gregory, S.T. and Knight, J.C., "A New Linguistic Approach to Backward Error Recovery", *Proc. FTCS-15*, 1985, pp.404-409.

[Hec76] Hecht, H., "Fault-Tolerant Software for Real-Time Applications", *Computing Surveys*, Dec. 1976, pp.391-407.

[Hor74] Horning, J.J. et al, " A Program Structure for Error Detection and Recovery", *Lecture Notes in Comp. Sci.*, Vol. 16, Springer-Verlag, 1974, pp. 171-187.

[Kim76] Kim, K.H., Russell, D.L., and Jenson, M.J., "Language Tools for Fault-Tolerant Programming", *PETP-1*, Electronic Sciences Lab., USC, Nov. 1976.

[Kim82] Kim, K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitor", *IEEE Trans. on Software Eng.*, Vol. SE-8, No. 3, May 1982, pp.189-197.

[Kim85] Kim, K.H., Yang, S.M., and Kim, M.H., "Implementation of Concurrent Programming Language Facilities Supporting Conversation Structuring", Proc. *COMPSAC 85*, Oct. 1985, pp.445-453.

[Kim89] Kim, K.H. and Yang, S.M., "Performance Impacts of Lookahead Execution in the Conversation Scheme", *IEEE Trans. on Computers*, Vol. 38, No. 8, August 1989, pp.1188-1202.

[Kol80] Kolstad, R.B. and Campbell, R.H., *Path Pascal User Manual*, Univ.of Illinois at Champaign-Urbana, Jan. 1980.

[Man89] Mancini, L.V., and Shrivastava, S.K., "Replication within Atomic Actions and Conversations: A Case Study in Fault-Tolerance Duality", *Proc. FTCS-19*, 1989, pp.454-461.

[Oza88] Ozake, B.M., Fernandez, E.B., and Gudes, E., "Software Fault Tolerance in Architectures with Hierachical Protection Levels", *IEEE Micro*, Vol. 8, Aug. 1988, pp.30-43.

[Ran75] Randell, B., "System Structure for Software Fault Tolerance", *IEEE Trans. on Software Eng.*, June 1975, pp.220-232.

[Rus79] Russell, D.L. and Tiedeman, M.J., "Multiprocess Recovery using Conversations", *Proc. FTCS-9*, 1979, pp.106-109.

[Shi87] Shirivastava, S.K., Mancini, L., and Randell, B., "On the Duality of Fault Tolerant System Structures", *Tech. Memo. SRM/455*, Computing Lab., Univ. of Newcastle upon Tyne, 1987.

[Toy87] Toy, W.N., "Fault-Tolerant Computing", A chapter in *Advances in Computers*, Vol. 26, Academic Press, 1987, pp.201-279.

[Tyr86] Tyrrell, A.M., and Holding, D.J., "Design of Reliable Software in Distributed Systems Using The Conversation Scheme", *IEEE Trans. on Software Eng.*, Vol. SE-12, No. 9, Sept. 1986, pp.921-928.

[Wu89] Wu, J. and Fernandez, E.B., "A Simplification of a Conversation Design Using Petri Nets", *IEEE Trans. on Software Engineering*, Vol. 15, No. 5, May 1989, pp.658-660.

Appendix A.VIII

Programmer-Transparent Coordination of Recovering Parallel Processes: Philosophy and Rules for Efficient Implementation

Programmer-Transparent Coordination of Recovering Concurrent Processes: Philosophy and Rules for Efficient Implementation

K. H. KIM, SENIOR MEMBER, IEEE

Abstract—A new approach to coordination of cooperating concurrent processes, each capable of error detection and recovery, is presented. Error detection, rollback, and retry in a process are specified by a well-structured language construct called recovery block. Recovery points of processes must be properly coordinated to prevent a disastrous avalanche of process rollbacks. In contrast to the previously studied approaches that require the program designer to coordinate the recovery block structures of interacting processes (thereby coordinating the recovery points of processes), the new approach relieves the program designer of that burden. It instead relies upon an intelligent processor system (that runs processes) capable of establishing and discarding the recovery points of interacting processes in a well coordinated manner such that 1) a process never makes two consecutive rollbacks without making a retry between the two, and 2) every process rollback becomes a minimum-distance rollback. Following the discussion of the underlying philosophy of the new approach, basic rules of reducing storage and time overhead in such a processor system are discussed. Throughout this paper examples are drawn from the systems in which processes communicate through monitors.

Index Terms—Error recovery, monitor, process interaction, programmer-transparent coordination, recovery block, rollback propagation.

I. INTRODUCTION

ROLLBACK and retry is a technique of saving the state of computation at various points during program execution and, if an error is detected, reestablishing the previously saved state of computation and restarting from that point [4], [16]. It is also called a backward recovery technique. The point in execution when the state of computation is saved is called a *recovery point* (RP) and thus "establishment of an RP" implies the saving of a computation state. In applications requiring high reliability, it has been a frequent practice to incorporate rollback and retry capability into systems in one form or another [7], [13]–[15].

When a system is designed as a collection of cooperating concurrent processes [5], [6] and each process is capable of rollback and retry, rollback of a process may cause rollbacks of other processes that interacted with the process [14], [17], [18]. For example, suppose that after exchanging information with process Y, process X has detected an error and rolls back to an RP $X.r$ established before the information exchange. The information that process X sent to process Y during the exchange is then *revoked* and process Y must also roll back to an RP established before receiving the revoked information. Since process rollback may propagate further, rollback of a process in some systems could initiate a disastrous avalanche of rollback propagations between processes, termed a *domino effect* by Randell [14]. One approach to preventing the domino effect is to require the program designer to carefully structure programs (executed by concurrent processes) such that interacting processes establish RP's in a well-coordinated manner [12], [14]. The program designer must ensure that the length of the longest possible chain of rollback propagations cannot exceed the tolerable limit. This could often become an unbearable (or undesirable) burden on the program designer.

A different approach is explored in this paper. The new approach called the *programmer-transparent coordination* (PTC) scheme here relieves the program designer of the burden of coordinating the RP's of concurrent processes. This means that the program designer can structure error detection and recovery capability of a process independently of the recovery structures of other processes. The new approach relies upon an intelligent processor system that (runs processes and) automatically establishes some RP's of interacting processes, in addition to the RP's that the processes are explicitly designed to establish. The RP's are added such that 1) a process never makes two consecutive rollbacks without making a retry between the two, and 2) every process rollback becomes a minimum-distance rollback. Therefore, the new approach simplifies the program designer's task at the cost of increased overhead in a processor system. Basic rules of reducing storage and time overhead in such a processor system, including a rule for establishing the minimum number of RP's, are presented in this paper.

To make the discussion specific and present the approach in a sufficiently general form, it is assumed that error detection, rollback, and retry in each process are structured by a language construct developed by Horning

Manuscript received October 31, 1985. This work was supported in part by the Office of Naval Research under Contract N100014-87-K-0231, by the National Science Foundation under Grants MCS-80-12906 and INT-8796259, by the U.S. Army under Contract DASG60-79-C-0074, and by the U.S. Air Force under Contract F04701-77-C-0120

The author is with the Computer Engineering Program, Department of Electrical Engineering, University of California, Irvine, CA 92717.

IEEE Log Number 8820976.

Programmer-Transparent Coordination of Recovering Concurrent Processes: Philosophy and Rules for Efficient Implementation

K. H. KIM, SENIOR MEMBER, IEEE

Abstract—A new approach to coordination of cooperating concurrent processes, each capable of error detection and recovery, is presented. Error detection, rollback, and retry in a process are specified by a well-structured language construct called recovery block. Recovery points of processes must be properly coordinated to prevent a disastrous avalanche of process rollbacks. In contrast to the previously studied approaches that require the program designer to coordinate the recovery block structures of interacting processes (thereby coordinating the recovery points of processes), the new approach relieves the program designer of that burden. It instead relies upon an intelligent processor system (that runs processes) capable of establishing and discarding the recovery points of interacting processes in a well coordinated manner such that 1) a process never makes two consecutive rollbacks without making a retry between the two, and 2) every process rollback becomes a minimum-distance rollback. Following the discussion of the underlying philosophy of the new approach, basic rules of reducing storage and time overhead in such a processor system are discussed. Throughout this paper examples are drawn from the systems in which processes communicate through monitors.

Index Terms—Error recovery, monitor, process interaction, programmer-transparent coordination, recovery block, rollback propagation.

I. INTRODUCTION

ROLLBACK and retry is a technique of saving the state of computation at various points during program execution and, if an error is detected, reestablishing the previously saved state of computation and restarting from that point [4], [16]. It is also called a backward recovery technique. The point in execution when the state of computation is saved is called a *recovery point* (RP) and thus "establishment of an RP" implies the saving of a computation state. In applications requiring high reliability, it has been a frequent practice to incorporate rollback and retry capability into systems in one form or another [7], [13]–[15].

When a system is designed as a collection of cooperating concurrent processes [5], [6] and each process is capable of rollback and retry, rollback of a process may cause rollbacks of other processes that interacted with the process [14], [17], [18]. For example, suppose that after exchanging information with process Y, process X has detected an error and rolls back to an RP $X.r$ established before the information exchange. The information that process X sent to process Y during the exchange is then *revoked* and process Y must also roll back to an RP established before receiving the revoked information. Since process rollback may propagate further, rollback of a process in some systems could initiate a disastrous avalanche of rollback propagations between processes, termed a *domino effect* by Randell [14]. One approach to preventing the domino effect is to require the program designer to carefully structure programs (executed by concurrent processes) such that interacting processes establish RP's in a well-coordinated manner [12], [14]. The program designer must ensure that the length of the longest possible chain of rollback propagations cannot exceed the tolerable limit. This could often become an unbearable (or undesirable) burden on the program designer.

A different approach is explored in this paper. The new approach called the *programmer-transparent coordination* (PTC) scheme here relieves the program designer of the burden of coordinating the RP's of concurrent processes. This means that the program designer can structure error detection and recovery capability of a process independently of the recovery structures of other processes. The new approach relies upon an intelligent processor system that (runs processes and) automatically establishes some RP's of interacting processes, in addition to the RP's that the processes are explicitly designed to establish. The RP's are added such that 1) a process never makes two consecutive rollbacks without making a retry between the two, and 2) every process rollback becomes a minimum-distance rollback. Therefore, the new approach simplifies the program designer's task at the cost of increased overhead in a processor system. Basic rules of reducing storage and time overhead in such a processor system, including a rule for establishing the minimum number of RP's, are presented in this paper.

To make the discussion specific and present the approach in a sufficiently general form, it is assumed that error detection, rollback, and retry in each process are structured by a language construct developed by Horning

Manuscript received October 31, 1985. This work was supported in part by the Office of Naval Research under Contract N100014-87-K-0231, by the National Science Foundation under Grants MCS-80-12906 and INT-8796259, by the U S Army under Contract DASG60-79-C-0074, and by the U.S. Air Force under Contract F04701-77-C-0120.

The author is with the Computer Engineering Program, Department of Electrical Engineering, University of California, Irvine, CA 92717.

IEEE Log Number 8820976.

et al., called the *recovery block* (RB) [9], [19]. For the same reason, it is also assumed that processes communicate through *monitors* [3], [6], [8]. A short review of the basic characteristics of an RB and a monitor is given below.

The RB has the following syntactic structure: **ensure** T **by** B_1 **else-by** B_2 **else-by** $\cdots$ **else-by** B_n **else-error**, where T denotes the *acceptance test*, B_1 the *primary try block*, and B_k $(2 \leq k \leq n)$ the *alternate try blocks*. All the try blocks in an RB specify computations aimed at producing the same or approximately the same result. A process executes the acceptance test T on exit from a try block to confirm that the result of the try block execution is acceptable. If it is acceptable, the process exits from the RB. If not, the process enters the next alternate try block. Also, the process enters the next try block if the underlying processor system detects an error (e.g., divide-by-zero) while the process is inside a try block.

Before an alternate try block is entered, the process state is restored to the state that existed just before entry to the primary try block. That is, the process rolls back to the RP established on entry to the RB. Each variable that was assigned a new value by the rejected execution is restored to its original value. The underlying processor system automatically performs this "assignment reversal." To enable this, the processor system can save, on initiation of an *RB execution* (RBE), all the variables that may be modified inside the RB. More efficient schemes were developed in [2], [10]. When an RBE is successfully completed, the RP established on initiation of the RBE may be discarded. This means that the process (which has successfully completed the RBE) discards the records of original values of the variables that have been kept to enable rollback to the entry of the RB.

The monitor which is the sole interprocess communication mechanism assumed in this discussion, consists of a shared data structure and all the operations, called *monitor procedures*, that processes can perform on it. No more than one monitor procedure of the same monitor may be in execution at any instant. A process X which has gained exclusive possession of a monitor and entered one of the monitor procedures, can release the monitor in two ways. one is to exit from the monitor procedure and the other is to execute a "wait(q)" where q is the name of a queue, called a *condition queue*, into which process X will enter to sleep. Without loss of generality, the capacity of each condition queue is assumed to be one [3]. A process Y which enters a monitor procedure while another process X is sleeping in condition queue q may waken the process X by executing a "signal(q)." A process can execute this signal operation only as it exits a monitor procedure. A process checks, at various stages during execution of a monitor procedure, to see if the current state of the monitor satisfies the prerequisite condition for the next operation, if the condition is not satisfied, the process will enter a condition queue.

The basic principles of the PTC scheme related to coordination of both the RP establishments and the rollbacks of interacting processes are valid independently of both the scheme of designing rollback and retry capabilities into processes and the mechanism by which processes communicate. Although the monitor-based formulation of the PTC scheme is presented in this paper, it is not considered a complex task to convert the formulation into one based on the use of a message-passing mechanism for interprocess communication. Nevertheless, the formulation provided here should help visualizing implementations of the scheme that are appropriate for use in centralized multiprogramming systems or microcomputer networks in which processes running on microcomputers communicate via monitors implemented on shared memory modules (e.g., Fig. 1).

In Section II, basic problems arising in communication among fault-tolerant concurrent processes are delineated and then the underlying philosophy of the PTC approach is discussed. Section III discusses basic rules for practical realization of the approach. This paper presents only the basic rules on the basis of which various efficient implementations can be devised. An example of an efficient implementation scheme based on the rules is described in [1] and [11].

II. A Programmer-Transparent Approach to Coordination of Recovering Concurrent Processes

For convenience in exposition and specific illustration of basic principles, the systems of processes dealt with in the rest of this paper are assumed to have the following characteristics. For ease of reference, the assumptions, the notations, and the definitions that are developed in this paper are numbered A1, A2, etc., N1, N2, etc., and D1, D2, etc., respectively.

Assumptions:

(A1) All the processes are created during system initialization and, once created, either exist forever or terminate together.

(A2) Processes can communicate only through monitors, and every resource shared among processes must be contained within a monitor.

(A3) There must be both a procedure (known to the underlying processor system) by which the state of a monitor can be recorded at any instant and a procedure by which a recorded state can be restored later.

(A4) Monitor procedures do not contain wait or signal instructions. (This assumption is removed in Section III-D.)

Assumption A3 is trivially met if the contents of shared variables (including condition queues) in a monitor completely represent the state of the monitor, because at any time the contents can be readily copied into another memory region and later restored, if necessary. Recording and restoring a state of a special resource (e.g., disk) contained in a monitor may require special procedures unique to the resource, perhaps supplied by the program designer. This then would be the only burden put on the program designer. A trivially simple example of a moni-

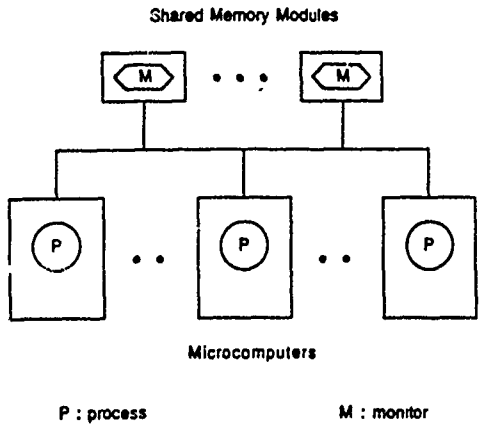


Fig. 1. An example microcomputer network.

tor satisfying assumption A4 (and A3) is shown in Fig. 2(a). The monitor LCLOCK consists of two shared integer variables (HOUR and MINUTE) and three monitor procedures that can operate on the shared variables. No more than one of these three procedures can be in execution at any instant.

Fig. 2(b) depicts a system of two processes communicating through monitor LCLOCK, and a segment of its execution history is shown in Fig. 2(c). Fig. 2(c) uses the following notations.

Notation:

(N1) The processes progress downward.

(N2) A bracket represents an RB execution (RBE) and each short wavy line represents a recovery point (RP) of a process. The bottom end of a bracket represents a *test point* (TP), i.e., an execution of the associated acceptance test.

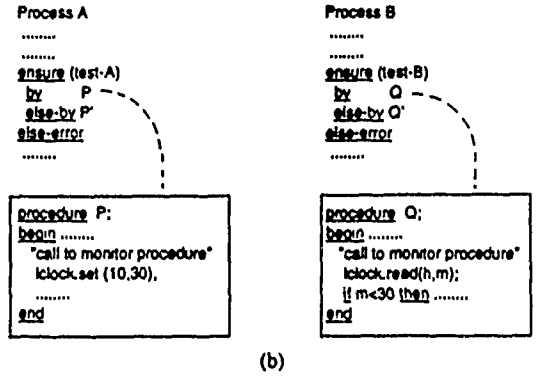
(N3) A shaded column represents the history of the state of a monitor, and a change in the direction of shading represents a state change.

(N4) A horizontal line represents execution of a monitor procedure. A line directed toward a shaded column represents a *monitor update operation* (i.e., execution of a monitor procedure which only changes the state of a monitor without examining the state at all), while a line directed toward a process from a shaded column represents a *monitor reference operation* (i.e., execution of a monitor procedure which does not change but examines the state of a monitor). In most cases a monitor procedure contains both reference and update operations. Execution of such a procedure is treated as a monitor reference operation immediately followed by a monitor update operation.

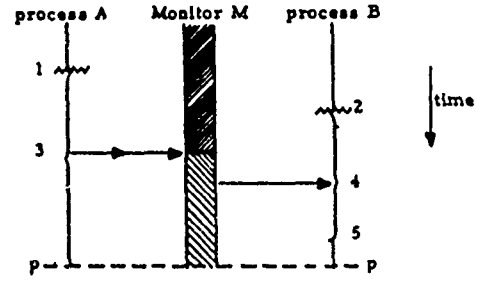
Process A in Fig. 2(c) produces information at $A.3$ for other processes and stores into monitor M [representing LCLOCK in Fig. 2(a)]. A little later process B picks up this information (or a part of it) at $B.4$. Process B passes its acceptance test at $B.5$. If process A fails at $A.p$, then it will roll back to RP $A.1$ and revoke the information it sent out between $A.1$ and $A.p$. (As part of this rollback, process A must restore monitor M to the state that existed prior to $A.3$.) Revocation of $A.3$ should cause process B

```
lclock: monitor
    var hour,minute: integer;
    monitor procedure set(var currenthour,currentmin: integer);
        begin hour := currenthour; minute := currentmin end;
    monitor procedure read(var currenthour,currentmin: integer);
        begin currenthour := hour; currentmin := minute end.
    monitor procedure update(var increment: integer);
        var temp: integer;
        begin temp := minute + increment;
          if temp<60 then minute := temp
          else begin minute := temp - 60;
                  if hour=23 then hour := 0
                             else hour := hour + 1
               end
        end "update;
    begin "initialization" hour := 0; minute := 0 end
end-monitor "lclock"
```

(a)



(b)



(c)

Fig. 2. (a) Monitor LCLOCK. (b) Two processes communicating through monitor LCLOCK (c) An execution history of the system in (b).

to roll back to an RP established prior to $B.4$, even though B has already passed its own acceptance test. A programmed acceptance test cannot detect all possible errors. Therefore, if process A fails, process rollback is propagated to process B *due to the revocation of the information sent out* by A. This type of rollback propagation is called an *R-propagation* in this paper.

On the other hand, if process B in Fig. 2(c) fails at $B.5$ and rolls back to RP $B.2$, then the question arises as to whether process A should roll back. The information received at $B.4$ may have caused the failure of B. Therefore, process B may be suspicious of the integrity of process A and ask A to roll back to an RP prior to $A.p$. This type of rollback propagation is *due to the suspicion* by process B and called an *S-propagation*.

When S-propagations are permitted, an avalanche of rollback propagations may occur unless the program de-

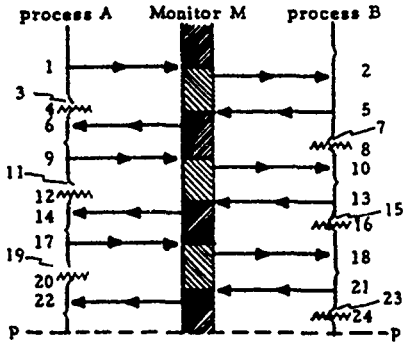


Fig. 3. A system history.

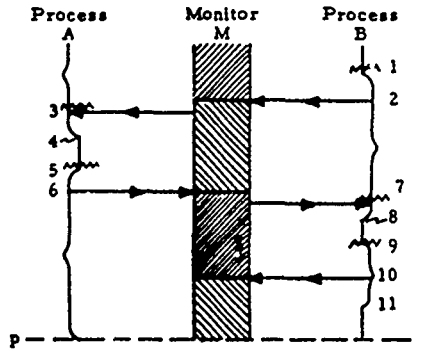


Fig. 4. Establishment of two branch-RP's *A*.3 and *B*.7.

signer takes great care in coordinating the RB structures of interacting processes. For example, if process *A* in Fig. 3 fails at *A.p*, then process *A* may be suspicious of the integrity of the information received at *A*.22. Process *B* may have supplied the information either in full at *B*.21 or in parts at *B*.5, *B*.13, and *B*.21. Upon receiving a request from process *A* to roll back to an RP preceding *B*.21, process *B* may in turn be suspicious of the integrity of the information received at *B*.18 and thus request process *A* to roll back to an RP preceding *A*.17. Continuing this way, both processes will have to roll back all the way to their beginnings (a domino effect occurs). Note that when S-propagations are permitted, a process can roll back to an RP *r* and then continue to roll back to another RP without making a retry at *r*. Because of this, S-propagations are prohibited in the PTC approach explored in this paper.

Allowing only R-propagations means that a process which sends out incorrect information is solely responsible for detection and correction of this error. Under this strategy a process can discard the RP established on initiation of an RB on passing the associated acceptance test if the process did not receive information from other processes but only sent out information during execution of the RB. For example, process *A* in Fig. 2(c) which does not know the speed of the progress of process *B* can discard RP *A*.1 on passing acceptance test *A.p*. In a sense, process *A* gives *a posteriori* accreditation of the information sent out between *A*.1 and *A.p* on passing *A.p* and then no other processes can challenge the integrity of the information.

When only R-propagations are permitted, it is possible to prevent a process from making two consecutive rollbacks (without a retry after the first rollback), by establishing RP's immediately before certain, but not all, monitor reference operations. These RP's, are in addition to the RP's established on initiation of RB executions (RBE's). The RP's established immediately before monitor references are called *branch-RP's*, while the RP's established on initiation of RBE's are called *base-RP's*. Establishment of branch-RP's is transparent to the program designer. For example, process *B* in Fig. 4 established a branch-RP immediately before monitor reference *B*.7. Such an RP will be referred to by the name (e.g., *B*.7) of the immediately following monitor operation in the rest of this paper. Process *A* also established branch-RP *A*.3. If process *A* fails acceptance test *A.p* and revokes monitor update *A*.6, then process *B* rolls back to RP *B*.7 and no more rollback propagation will ensue. In this case, process *B* makes a *minimum-distance rollback*. If branch-RP's *B*.7 and *A*.3 had not been established, process *B* would have rolled back to base-RP *B*.1 and revoked monitor update *B*.2, which in turn would have caused process *A* to roll back to an RP preceding *A*.3; process *A* would thus have made two consecutive rollbacks, first to RP *A*.5 and then to the RP preceding *A*.3, without a retry from *A*.5.

Fig. 4 also reveals an additional aspect of monitor update accreditation. Before process *A* executes acceptance test *A.p*, process *B* has passed acceptance test *B*.11 and thus has given *a posteriori* accreditation of monitor update *B*.10. Note that this accreditation is indirectly voided when process *A* fails at *A.p* and causes process *B* to roll Back to *B*.7, although the accreditation issued is not directly challenged. This is reasonable because the integrity of process *B* has been in a questionable state since receiving uncommitted information at *B*.7.

In short, the essence of the PTC approach is to prohibit S-propagations and to establish certain branch-RP's in addition to base-RP's, thereby ensuring that a process never makes two consecutive rollbacks. A process need also establish RP's of a monitor by saving the states of the monitor immediately before certain, but not all, monitor update operations in order to be able to restore the monitor if the process has to revoke the updates later. All the management functions, including maintenance of RP's, restoration of monitors, and coordination of process rollbacks, must be automatically handled by the underlying processor system. The PTC approach supports structuring of the error detection and recovery capabilities of each process in a manner independent of those of other processes. It also supports an "optimistic" process, i.e., a process sending to other processes revocable information which has not been completely validated The approach is practical only if at all times the number of RP's of processes and the number of RP's of monitors that need to be maintained are manageable Basic rules for efficient implementation of the PTC approach are discussed in the next section.

III. Rules for Efficient Implementation of a Processor System Capable of Coordination of Recovering Concurrent Processes

It was mentioned in the preceding section that the feasibility of the PTC approach is largely dependent upon the reduction of the time and storage overhead in an intelligent processor system. Two major sources of overhead are in creation and discard of 1) RP's of processes and 2) RP's of monitors. Three basic rules for reduction of this overhead are now discussed. To simplify the presentation, Sections III-A, III-B, and III-C introduce the rules and illustrate them under the restrictive assumption A4 and the following.

Assumption (A5) Processes communicate through a single monitor. (This assumption is removed in Section III-E).

Then Sections III-D and III-E show that the rules are applicable to general cases where the restrictive assumptions A4 and A5 are not made.

A. Rule of Minimal Recovery Point (RP) Maintenance

To get an intuitive idea, consider Fig. 5. Process B established a branch-RP at the beginning of monitor reference $B.4$ since the information existing then in the monitor was subject to possible future revocation (due to the possible failure of the RBE initiated at $A.1$). However, it was not necessary for process B to establish additional RP's at monitor references between $B.6$ and $B.j$. This is because any information picked up from M between $B.4$ and $B.j$ is revoked only when process A rolls back to $A.1$, and thus the references (made between $B.6$ and $B.j$) are voided together with reference $B.4$. Therefore, an ongoing RBE of a process X causes another process Y to create at most one branch-RP, and process Y must maintain the RP at least until the RBE (of X) deceases. If process A in Fig. 5 passes acceptance test $A.p$, then RP $B.4$ may be discarded. (Communication of validation results and other notices among processes can be implemented in various ways and is not dealt with in this paper. One implementation is described in [11].) Base-RP $B.2$ can not be discarded even after acceptance test $B.k$ succeeds as long as branch-RP $B.4$ remains. This is because once process B rolls back to branch-RP $B.4$, it can fail at $B.k$ in which case it needs to roll back to base-RP $B.2$. In a sense, the RBE that was initiated at $B.2$ is not completely validated even after $B.k$ until the RBE that was initiated at $A.1$ is completely validated at $A.p$ and branch-RP $B.4$ is discarded.

In order that the conditions for establishment and discard of RP's can be stated precisely, the following notations and terms are introduced.

Notation (N5) An RBE is represented by $[a:]$ or $[:b]$, where a and b are the starting and the ending execution points, respectively, of the RBE. For example, $[B.2:]$ and $[:B.k]$ in Fig. 5 represent the same RBE.

Definitions:

(D1) When a monitor M is updated by a process X during an RBE $[X.a:]$, the RBE $[X.a:]$ becomes a *direct potential recaller (DPR) of monitor M* and holds that right until $[X.a:]$ deceases. For example, $[A.1:]$ in Fig. 5 is a DPR of M from $A.3$ to $A.p$. Monitor M is said to be *rollback-chained* or *R-chained* to the base-RP established at the starting point $X.a$ of RBE $[X.a:]$.

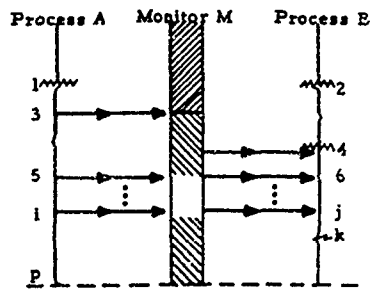


Fig. 5. A system history.

(D2) When a monitor M has no DPR's, it is said to be *rollback-free*.

(D3) If a process Y references a monitor M which is R-chained to base-RP $X.a$, then processs Y becomes *R-chained to RP X.a through monitor M*, and thus RBE $[X.a:]$ becomes a *DPR of process Y*. If process Y becomes R-chained to RP $X.a$ during an RBE $[Y.c:]$, then RBE $[Y.c:]$ is said to be R-chained to RP $X.a$. Thus RBE $[X.a:]$ is a *DPR of RBE* $[Y.c:]$

(D4) An RBE of a process X is a *DPR of process X itself*.

For example, RBE $[A.2:]$ in Fig. 6 becomes a DPR of M at its first update (A.3) of M, and RBE $[B.1:]$ of process B becomes R-chained to base-RP $A.2$ at the first reference ($B.4$) to M after M became R-chained to the RP $A.2$. Monitor M has two DPR's $[A.2:]$ and $[B.1:]$ immediately after $B.5$. By definition D4, RBE $[B.1:]$ is a DPR of process B between $B.1$ and $B.p$.

Definitions:

(D5) The set of all the DPR's of a process X (or a monitor M) at a given time t is called the *direct potential recaller set (DPRS)* of X (or M) at t, and is denoted by DPRS(X, t) [or DPRS(M, t)]. The second argument t may be omitted if there is no ambiguity. Similarly, the DPRS of an RBE $[X.a:]$ at t is defined and denoted by DPRS$([X.a:], t)$, where t may be omitted if there is no ambiguity.

(D6) The *potential recaller closure (PR*)* of an RBE $[X.a:]$ at a given instant, denoted by PR*$([X.a:])$, is a set of RBE's defined as follows: every DPR of RBE $[X.a:]$ is a member of PR*$([X.a:])$; if an RBE e is a member of PR*$([X.a:])$, then every DPR of e is also a member of PR*$([X.a:])$.

(D7) An RBE $[Z.e:]$ which is not a DPR of RBE $[X.a:]$ but a member of PR* $([X.a:])$, is called an *indirect potential recaller (IPR)* of RBE $[X.a]$.

(D8) The set of all the IPR's of an RBE $[X.a:]$ is called the *indirect potential recaller set (IPRS)* of RBE $[X.a:]$ and is denoted by IPRS $([X.a:])$.

The DPRS of an RBE $[X.a:]$ can be a proper subset of PR*$([X.a:])$ because PR*$([X.a:])$ may increase after

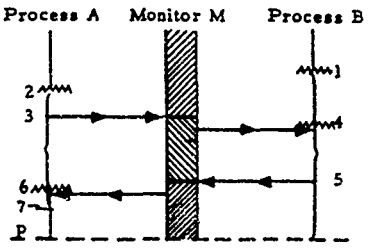


Fig. 6. A system history.

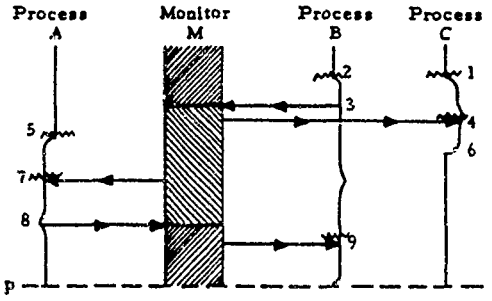


Fig. 7. An indirect potential recaller [A.5:] of RBE [C.1:].

process X passes the acceptance test of RBE [$X.a$:] (and thus DPRS([$X.a$:]) is fixed). For example, process C in Fig. 7 already passed the acceptance test of RBE [$C.1$:] at $C.6$. Thus DPRS([$C.1$:]) was fixed at $C.6$ and included ongoing RBE [$B.2$:] of process B. When RBE [$B.2$:] became R-chained to RP $A.5$ of another process A later at $B.9$, RBE [$A.5$:] became a member of PR*([$C.1$:]) but not a DPR of [$C.1$:].

Definitions:

(D9) An RBE [:$X.b$] is said to be *partially validated* if acceptance test $X.b$ has been successful but there remain DPR's of [:$X.b$] (i.e., there are RBE's of other processes which became DPR's of process X during [:$X.b$] and have not deceased).

(D10) An RBE [:$X.b$] is *completely validated* if 1) acceptance test $X.b$ has been successful and 2) either PR*([:$X.b$]) is empty or every member of PR* ([:$X.b$]) has been completed with passing of the associated acceptance test.

An informal definition of "complete validation" is validation of every aspect of an RBE. Therefore, once an RBE [$X.a$:] is completely validated, there will never be any need for process X to roll back to RP $X.a$. In the case of Fig. 6, complete validation of RBE [$A.2$:] or [$B.1$:] consists of acceptance tests $A.7$ and $B.p$. RBE [$A.2$:] is first partially validated at $A.7$. Then RBE [$A.2$:] becomes completely validated together with [$B.1$:] at $B.p$ when every member of PR*([$A.2$:]) = PR*([$B.1$:]) = {[$A.2$:], [$B.1$:]} has obtained the partially validated status On the other hand, RBE [$C.1$:] in Fig. 7 is not completely validated even when acceptance test $B.p$ succeeds, because PR*([$C.1$.]) still contains ongoing RBE [$A.5$.] Fig. 8 depicts various states that a newly initiated RBE may go through.

The rule for maintaining the DPRS's of processes and

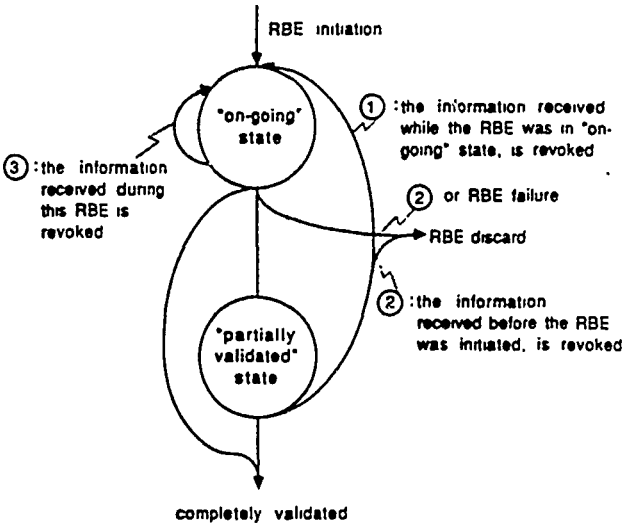


Fig. 8. Life cycle of an RBE.

of the monitor stems from definitions D1–D5, and is stated below.

Rule (R1) "DPRS management".

(R1.1) RBE Initiation: When a process X is about to initiate an RBE at $X.a$, DPRS(X) is updated to DPRS(X) ∪ {[$X.a$:]} where ∪ is a set-union operator.

(R1.2) Monitor Update: When a process X is about to update a monitor M, DPRS(M) is updated to DPRS(M) ∪ DPRS(X).

(R1.3) Monitor Reference: When a process X is about to reference a monitor M, DPRS(X) is updated to DPRS(X) ∪ DPRS(M).

(R1.4) Complete Validation of an RBE: If an RBE [$X.a$:] has been completely validated, [$X.a$:] is removed from the DPRS's of processes and from the DPRS of the monitor.

(R1.5) Discard of an RBE: If an RBE [:$X.b$] is discarded due to a failure at or before $X.b$, [:$X.b$] is removed from the DPRS's of processes and from the DPRS of the monitor.

By using the DPRS's and IPRS's of processes and the monitor, a rule for correctly establishing RP's can be stated as follows.

Rule (R2) "Minimal RP maintenance".

(R2.1) RP Establishment: When the DPRS of process X, DPRS(X), is expanded by incorporating a set E of new members, a new RP is established by X. E is called the *DPRS of the RP*.

(R2.2) RP Discard: An RP may be discarded when all the members of its DPRS E have been removed from DPRS(X).

Note that the DPRS's of RP's are mutally disjoint and that *the DPRS of a process X is the same as the union of the DPRS's of all the RP's maintained by X* at that time. In addition, *the DPRS of RBE* [$X.b$] *at X.b is equal to the union of the DPRS's that were established during* [$X.b$] *and have remained* As an illustration of the above rule, DPRS(A) in Fig. 5 is expanded at $A.1$ and thus a base RP is established at $A.1$. Similarly, DPRS(B) is ex-

panded at $B.4$ and thus a branch-RP is established at $B.4$. When acceptance test $A.p$ succeeds, RBE $[A.1:]$ is completely validated and thus is removed from DPRS(A), DPRS(B), and DPRS(M). Since the DPRS's of both RP's $A.1$ and $B.4$ are now empty, the two RP's are discarded. Also, RBE $[B.2:]$ has now become completely validated and RP $B.2$ is discarded. In the case of Fig. 4, RP $B.9$ is discarded when acceptance test $B.11$ succeeds but RP's $B.7$, $B.1$, $A.5$, and $A.3$ are all discarded when acceptance test $A.p$ succeeds.

The following statement can be made about this RP management rule.

Lemma (L1):

Under rule R2 processes establish the minimum number of RP's required to enable every rollback to be made to the most recent valid execution point. Moreover, if RBE's can be removed from the DPRS's as soon as they are completely validated or discarded, no redundant RP's are kept except where cyclic dependency develops among a set of RBE's. In the latter case, at most one redundant RP may be kept per cycle of dependent RBE's.

Proof: To each current DPR e of a process X there corresponds exactly one valid RP $X.r$ established by the process X when RBE e became a DPR of X. The RP $X.r$ is obviously the most recent valid execution point of the process X when RBE e has failed. On the other hand, to each RP $X.r$ currently maintained by process X there correspond one or more active DPR's of the process X. Since the process X rolls back when and only when any of its DPR's fails, the process in general maintains no redundant RP's. However, in a situation where cyclic dependency develops among a set of RBE's, a branch-RP may be kept longer than the minimum required period. For example, consider the case of Fig. 6 in which RBE's $[A.2:]$ and $[B.1:]$ become mutually dependent. Under rule R2 RP $B.4$ is discarded at $B.p$ when the DPR of the RP, i.e., $[A.2:]$ becomes completely validated. However, RP $B.4$ may be discarded at $A.7$ when the DPR of the RP becomes partially validated. This is because the partially validated RBE $[A.2:]$ can roll back only when RBE $[B.1:]$ fails in which case execution point $B.4$ is automatically cancelled out as a part of the rollback to base-RP $B.1$. Only the branch-RP established within the last finishing RBE among the RBE's in a cycle may become a redundant RP. This is because the branch-RP's established within other RBE's in the cycle may become needed as a result of the failure of the last finishing RBE. When each of those branch-RP's is used for a rollback, its immediate cause is not the failure of the RBE that encompasses the RP. Therefore, at most one such redundant RP may be kept per cycle of dependent RBE's. Q.E.D.

An example of an RP having more than one DPR is RP $B.7$ in Fig. 9 which has two DPR's, $[A.1:]$ and $[A.5:]$, until $A.10$. This means that RP $B.7$ must remain intact until $A.11$ (even after $[A.5:]$ is successfully completed at $A.10$). Therefore, the number of RP's maintained by a process X is always less than or equal to the cardinality of DPRS(X). DPRS(X) is of course a subset of ongoing

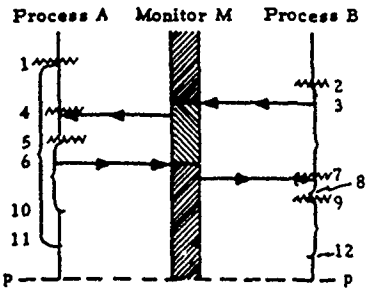


Fig. 9. A branch-RP ($B.7$) having two direct potential recallers.

or partially validated RBE's of the processes in the system.

B. Rule of Reducing the Number of RP's

Although minimal sets of RP's are maintained under the RP management rule R2 given in the preceding section, the number of RP's maintained at one time could, in some cases, exceed the tolerable limit (imposed mainly by storage space available). For example, consider Fig. 10(a) which looks similar to but possesses the opposite characteristics of Fig. 3 from the RP management point of view. (Every RBE in Fig. 3 is completely validated as its associated acceptance test succeeds.) Each time an acceptance test succeeds, it results only in a partial validation because there is a DPR which is active at that time. Therefore, no RP's could have been discarded by $A.p$. If process A passes acceptance test $A.p$, then all the RP's (except $B.25$) can be discarded.

In order to avoid intolerable accumulation of partially validated RBE's and associated PR's in a process X, process X can safely remove old RP's which have low probability of being used. For example, RP $B.1$ or $B.7$ in Fig. 10(a) is considered to have a comparatively low probability of being used because it can be used only when all the subsequent RBE's (by both processes A and B) that have been partially validated were actually wrong. Therefore, process B may remove $B.7$ and thus sacrifice the capability of making minimum-distance rollback (i.e., rollback to $B.7$) in the case where RBE $[A.5:]$ is discarded. Process B must retain base-RP $B.1$ after discarding RP $B.7$. If RBE $[A.5:]$ is later completely validated, then RP $B.1$ can also be discarded. If $[A.5:]$ is discarded, then process B can immediately declare that RBE $[B.1:]$ has failed and then roll back to $B.1$ as if it had made a retry from $B.7$ and failed the acceptance test of $[B.1:]$. This will cause process A to roll back to $A.3$. Therefore, process B as well as process A can still be recovered as long as it maintains the oldest RP, although it sacrificed the ability to make a minimum-distance rollback every time. In other words, a process can trade increase of rollback distance (in case of less probable rollbacks) for reduction of RP's.

Definition (D11) A base- or branch-RP r is said to be *supported by* a base-RP q if r was established during RBE $[q:]$. For example, RP $B.7$ in Fig. 10(a) is supported by RP $B.1$.

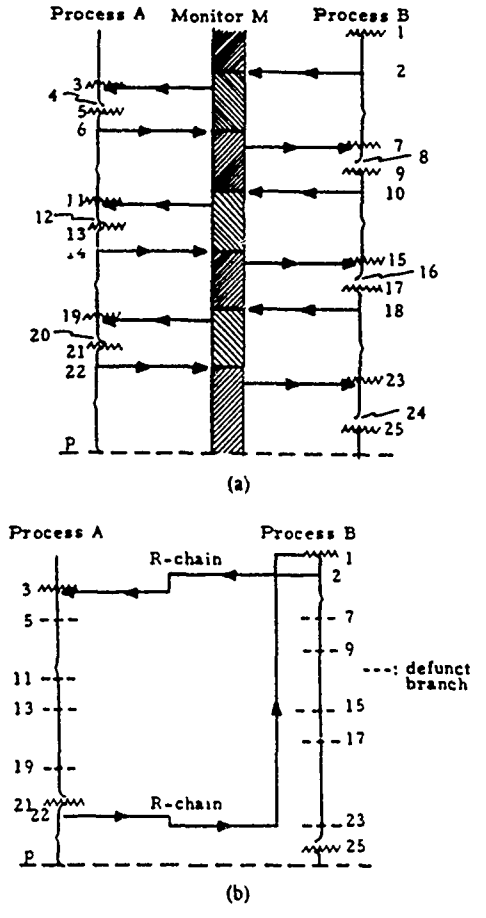


Fig. 10. (a) A system history. (b) Result after process B removes five RP's from (a).

A two-part rule for safe reduction of RP's is now given.

Rule (R3) "RP reduction".

(R3.1) A branch-RP r can be removed if 1) its immediately preceding RP q is a base-RP and 2) r is supported by q. Once removed, r is called a *defunct branch* of q. If process X is required to roll back to defunct branch r, it will immediately declare that RBE $[q:]$ has failed and then it will roll back to q.

(R3.2) A base-RP r can be removed if 1) its immediately preceding RP q is another base-RP and 2) there are no remaining RP's supported by r. Once removed, r and its defunct branches become defunct branches of q. When process X removes r, it notifies other processes and the monitor that the role of RBE $[r:]$ as a DPR is taken over by RBE $[q.]$. If process X is required to roll back to defunct branch r, it will immediately declare that RBE $[q:]$ has failed and then it will roll back to q.

The immediate effect of removing a branch-RP is local to the process. However, the effect of removing a base-RP may propagate to other processes. For example, suppose that process B in Fig. 10(a) first removed branch RP's $B.7$ and $B.15$ and then removed base-RP $B.9$. Now whenever process B is required to roll back to $B.9$, it has to roll back to $B.1$. This means that process A will never roll back to $A.11$, instead it will be required to roll back to $A.3$. In other words, the role of $[B.9.]$ as a DPR has been taken over by $[B.1.]$. Therefore, if a base-RP $X.j$ has been made a defunct branch of another base-RP $X.i$, RBE $[X.j:]$ must be removed from the DPRS's of processes and of the monitor. Removal of $[X.j:]$ from the DPRS of a process Y may enable discard of RP's in the process Y. Fig. 10(b) shows the result after process B removes the five RP's $B.7$, $B.15$, $B.9$, $B.23$, and $B.17$ in sequence. Again RP's $A.3$, $A.21$ and $B.1$ can be discarded when acceptance test $A.p$ succeeds.

C. Rule of Minimal Monitor Recovery Point Maintenance

We now turn to the problem of saving and restoring monitor states. See Fig. 11(a). When process A fails acceptance test $A.p$ and rolls back to $A.1$, process B rolls back to $B.4$. In addition, monitor M must be restored to the state that existed immediately before the first update $A.3$ by the failed RBE $[A.1:]$. In order to prepare for this monitor restoration, the responsible process A must establish an RP of M before it modifies M at $A.3$.

Actual saving of the monitor state may be done incrementally by use of the recovery cache scheme [2]. Each time a part of the monitor state variables needs to be modified, the original content of the part is saved into cache storage commonly accessible by all the processes that have rights of access to the monitor.

Notation (N6) A wavy line crossing a shaded column which represents the state history of a monitor M, represents an RP of M established by a process.

There is no need for a process to establish an RP of a monitor at the beginning of a reference to the monitor. To illustrate this, if process B in Fig. 11(a) failed acceptance test $B.6$ and rolled back to $B.2$ to make a retry, process A would not be aware of the rollback of B. Fig. 11(b) shows such a system history. As shown, process B makes a monitor reference ($B.4'$) again during the retry. Suppose monitor operation $A.5$ were a monitor reference unlike the case shown in the figure. Then the monitor state referenced at $B.4'$ (during the retry) would be identical to the state referenced at $B.4$ (during the previous unsuccessful try). In such case, saving the state of the monitor at the beginning of monitor reference $B.4$ was clearly unnecessary. On the other hand, if $A.5$ is a monitor update as shown, then either the information deposited at $A.5$ is not needed for reference $B.4'$ (or $B.4$) or reference $B.4$ was executed incorrectly. This is because of the asynchronism among processes in a system of concurrent processes. Each process must be designed to wait in a condition queue if the monitor does not contain the desired information. Suppose reference $B.4$ was a mistake and process B should have waited from $B.4$ until process A supplied the desired information at $A.5$. Now the monitor will contain all the desired information at reference B 4' during the entry. On the other hand, if $B.4$ was correct, i.e., the monitor contained all the desired information at B 4, then at monitor update $A.5$ process A would not know whether process B has already made a reference B 4 and thus would not destroy the information desired at $B.4$ (unless process A is faulty). Moreover, the information that process A

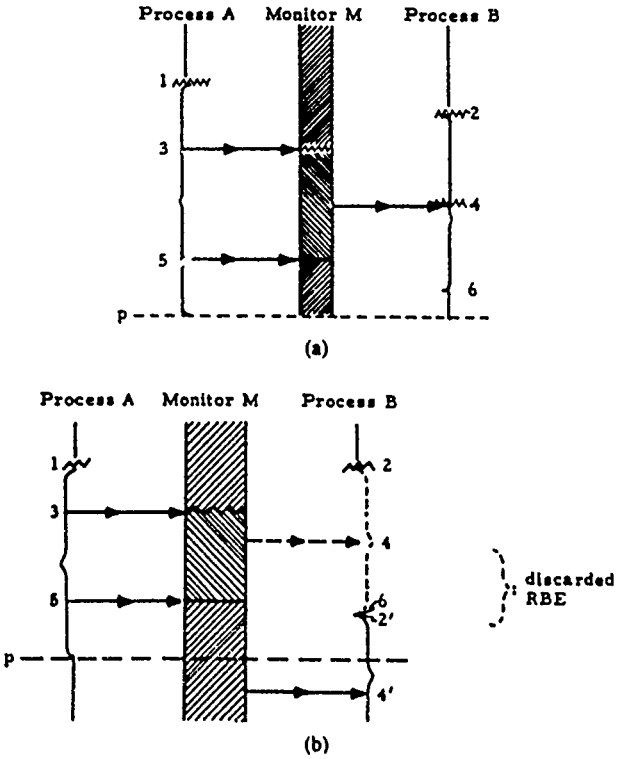


Fig. 11. (a) Establishment of a monitor RP (at $A.3$). (b)Monitor reference ($B.4$) not accompanying a monitor saving.

supplies at $A.5$ will not be needed at $B.4'$. Thus the monitor will again contain all the desired information at $B.4'$. Therefore, a monitor reference never accompanies establishment of a monitor RP.

Note in Fig 11(a) that a monitor RP is not established at the beginning of monitor update $A.5$. This is because the most recently established base-RP is A 1 and process A has already established a monitor RP at the beginning of $A.3$. In other words, any spontaneous rollback of process A occurring after $A.5$ will be made to base-RP $A.1$ (or to an earlier execution point) and thus will involve restoration of the monitor to the state that existed before the first update $A.3$ by RBE $[A.1:]$ which precedes $A.5$. The rule for establishing monitor RP's is stated below.

Rule (R4) "Minimal monitor RP maintenance": An RP of a monitor must be established when (and only when) the DPRS of the monitor is expanded by a set E of new members at a monitor update. Such an update is called an *R-chaining update* and E is called the *DPRS of the monitor RP*. The monitor RP may be discarded when all the members of E have been completely validated.

In systems using one monitor, every R-chaining update is the first update made by a process X during its RBE $[X.a.]$; in systems using multiple monitors, an update which is not the first to be made by a process during its RBE can be an R chaining update as will be shown in Section III-E.

Rollback of a monitor to one of its RP's has a unique aspect. Consider the case of Fig. 12. If process A fails at $A.p$ and rolls back to $A.1$, then it must also bring monitor M back to the monitor RP established at $A.2$ In doing so,

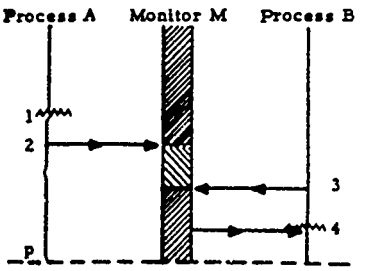


Fig. 12. A system history.

process A must undo only those update actions it has made since establishment of the monitor RP at $A.2$. Therefore, the part of the monitor updated by process B at $B.3$ but unmodified by process A since initiation of RBE $[A.1:]$ remains unaffected during the rollback of the monitor. In this way the useful information deposited by process B at $B.3$ is preserved through the rollback process. The part of the monitor updated by both processes A and B may be restored to the state that existed before $A.2$ since the relevant part of the information supplied at $B.3$ could have been supplied by process B before $A.2$ (due to the asynchronism among processes) and thus cannot be of critical nature. The above discussion implies that the monitor state must be saved in segments each representing the part modified by a process.

Another conceptually simpler approach to saving monitor states is conceivable. That is, a snapshot of the entire monitor state can be taken and saved. Each time a monitor needs to be brought back to an RP, the snapshot taken at the RP can be used to restore the monitor. This approach can be costly, however. For example, rollback of process A to $A.1$ in Fig. 12 causes rollback of process B to RP $B.4$ under rule R2 and the entire monitor state is restored to the state that existed before $A.2$. This means that the information produced at $B.3$ is completely lost. In order to prevent this kind of information loss, rule R2 should be modified such that an RP is established immediately before $B.3$ rather than $B.4$. In other words, every monitor update must be treated as a combination of a monitor reference and a monitor update with respect to RP management. This increases the cost of establishing and maintaining RP's. Since this approach trades the increase in the number of process RP's for the simplicity in saving monitor states, its cost-effectiveness depends on the interaction pattern of cooperating processes.

D. Handling of Wait and Signal Instructions

Assumption $A.4$ is removed now. A monitor procedure may involve one or more executions of a wait instruction and at most one execution of a signal instruction (at the end). Condition queues are now a legitimate part of the shared data structure, they must be included in monitor state saving and in monitor restoration. Three aspects of executing such a procedure which are not present when executing a monitor procedure without any wait or signal instructions are. 1) treating segments of a monitor procedure execution as atomic (uninterruptible) monitor op-

erations (with respect to both process interaction and R-chaining); 2) restoring condition queues as part of monitor restoration; 3) avoiding a system crash due to the entry of a process into a condition queue after depositing erroneous information into a monitor but before executing an acceptance test.

First, when a process executes a monitor procedure that has wait and signal instructions, it generally goes through an alternating sequence of monitor-possessing periods and waiting periods during which it does not possess the monitor. Only the process activity during each of those monitor-possessing periods is uninterruptible (by other processes) and thus is an atomic monitor operation. As before, an atomic monitor operation is classified as a reference operation, an update operation, or a reference-update operation. Since it is generally impossible to predetermine a sequence of atomic monitor operations involved in execution of such a monitor procedure, a practical approach is to classify the monitor procedure as a whole and then pass that classification to all of its atomic monitor operations as they are dynamically defined. Note that execution of a wait or signal instruction should be treated as an update action since it changes the content of a condition queue. Analogously, when a waiting processs is awakened, the awaking action is treated as a reference action since the awakened process has received information (i.e., wakening signal) from the signaling process. Under the above definition of an atomic monitor operation the three rules R2, R3, and R4 are valid without any modification.

Notation:

(N7) A waiting period of a process is represented by a gap in the vertical line that represents the progress of the process.

(N8) An undirected horizontal line represents a *monitor reference-update operation.*

For example, process A in Fig. 13 executes a monitor procedure (involving wait instructions) from $A.3$ to $A.8$. The monitor procedure execution consists of three atomic monitor reference-update operations ($A.3$, $A.6$, and $A.8$). Process A is in a condition queue after $A.3$ until the beginning of $A.6$ (at which time it is awakened by the signal generated by B at the end of $B.5$), and again after $A.6$ until the beginning of $A.8$. Note that process A establishes an RP at (the beginning of) $A.6$. Actually a process may establish multiple RP's during a single monitor procedure execution if the execution involves multiple atomic monitor operations.

Second, restoration of a monitor, which is required when a process rolls back to an RP, may include recalling some processes into the condition queues in which the processes slept previously. For example, assume that process B in Fig. 13 fails acceptance test $B.9$ and needs to roll back to $B.4$. Process B is then responsible for bringing the monitor back to the monitor RP that was established at monitor reference update $B.5$, which must include restoration of the condition queue that contained process A. On the other hand, restoration of a monitor

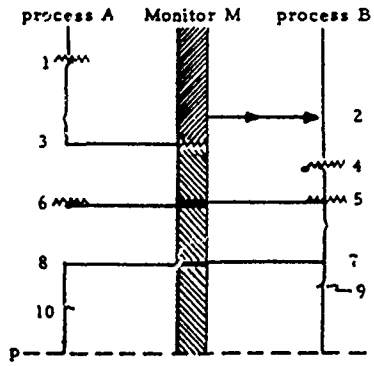


Fig. 13. A system history containing waiting periods of a process.

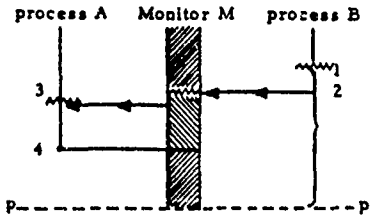


Fig. 14. A process (A) that needs to roll back out of a queue.

may also include driving processes out of some condition queues. For example, assume that process B in Fig. 14 has failed acceptance test $B.p$. At this point process A is sleeping in a condition queue and has to roll back out of the condition queue.

Third, a situation can occur in which a process that has produced erroneous information for other processes cannot have an opportunity to execute its own acceptance test. The systems would crash unless additional features have been incorporated. In Fig. 15, assume that process A stores erroneous information into the monitor store at $A.3$. Process B takes the erroneous information at $B.4$ and subsequently fails the acceptance test at $B.p$. A retry by process B might fail again at $B.p$ since the erroneous information generated at $A.3$ is still in the monitor and is referenced again. Meanwhile process A is not aware of the failure(s) of process B and is waiting inside a condition queue for a wakening signal that has to be supplied by process B (after $B.p$) but has not been produced because process B has not passed $B.p$. In a sense, this is a deadlock situation.

If process B does not maintain any RP earlier than $B.1$, then process B will have to be abandoned after the unsuccessful retries with all the available alternates and the system will crash. In this case, the system crashes despite the possibility that the acceptance test of RBE [$A.2$:] may be able to detect the error and a retry by process A from $A.2$ may circumvent the error. That is, the resources in the system are not fully utilized.

There are two possible solutions to this problem. One is to provide a special programmer-transparent "watch dog process" that is responsible for periodically examining the status of every process, upon detection of a process that has spent an excessive amount of time in a condition queue, the watchdog process forces the process

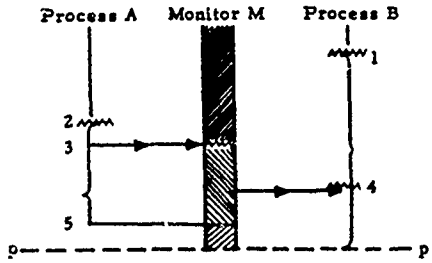


Fig. 15. A process (A) entering a condition queue after depositing erroneous information.

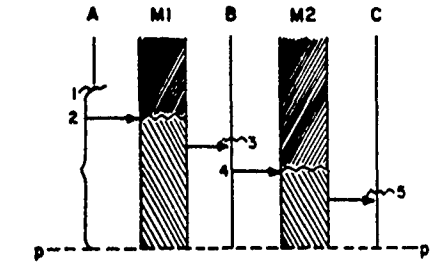


Fig. 16. A rollback chain connecting five system components.

to roll back to the immediately preceding base-RP and to begin a retry. A process X that has exhausted all the alternates waits (at the point of the last failure) until another process Y that interacted with X and has slept long, is forced by the watchdog process to roll back. Then process X initiates another retry. The other solution is to allow S-propagation as a last resort when process X has exhausted all the alternates and is about to be abandoned. That is, process X requests the processes that had produced information which was referenced by X, to roll back to their earlier base-RP's, and thereafter process X makes a retry. This exception (i.e., S-propagation) can be justified since the system would have crashed anyway. Either way, the error is indirectly detected and recovered.

E. Systems of Processes Communicating Through Multiple Monitors

. Assumption A5 is removed now. Processes may communicate through multiple monitors.

First, consider the systems containing multiple nonnested monitors. The rules R2, R3, and R4 are directly applicable to such systems. For example, process B in Fig. 16 became R-chained to RBE [A.1:] of process A at B.3 and thus established a branch-RP. At B.4 process B executed an update that R-chained monitor M2 to [A.1:] and thus caused an RP of M2 to be established. Process C then became R-chained to [A.1:] at C.5. Therefore, an R-chain has been established that connects RBE [A.1:] to process C (at C.5) through three other system components, M1, B, and M2. Rollback of process A to RP A.1 will require rollback of process B to RP B.3 which will in turn require rollback of process C to RP C.5.

Second, consider the systems in which monitors are nested, i.e., a monitor procedure may call other (nested) monitor procedures. Fig. 17(a) depicts such a system in which M3 is a nested monitor.

Execution of a monitor procedure calling other monitor procedures is treated as a compound monitor operation which may be broken into several atomic operations between which RP's may be established. For example, see Fig. 17(b) that shows an execution history of the system in Fig. 17(a). The figure uses the following notation.

Notation (N9) A continuous execution of a monitor procedure including calls to nested monitor procedures is represented by a vertically shaded horizontal column.

From A.3 to A.7, process A executes a monitor procedure that belongs to M1 and involves two calls to monitor procedures belonging to M3. Similarly, process C executes a compound monitor operation from C.2 to C.8. On the other hand, the monitor operation C.9 is an atomic monitor operation since the monitor procedure (of M2) executed does not involve either a call to M3 nor an execution of wait or signal instruction. For the same reason, every execution of a procedure of M3 shown in the figure is an atomic monitor operation.

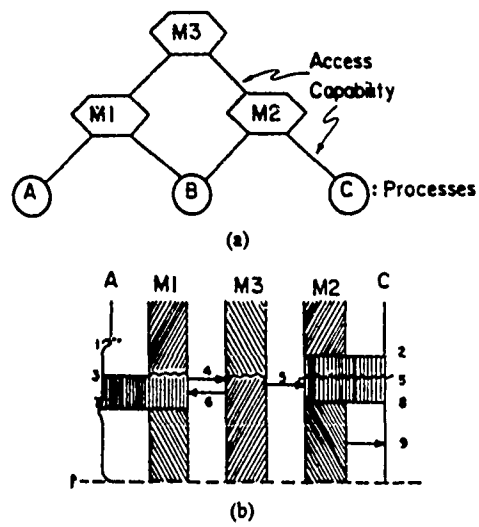


Fig. 17. (a) A system containing a nested monitor M3. (b) An execution history of the system in (a).

With the above definition of an atomic monitor operation the rules R2, R3, and R4 are directly applicable to the systems containing nested monitors as illustrated in Fig. 17(b). RBE [A.1:] became a DPR of M3 at A.4, and an RP of M3 was established at that time. Process C then became R-chained to RP A.1 at C.5 and thus an RP was established. This RP establishment was accompanied by establishment of a monitor RP of monitor M2. If process A fails acceptance test $A.p$, then it will restore both M3 and M1 to the states that existed at A.4 and A.3, respectively, and roll back to A.1. This will in turn cause process C to roll back to C.5, which includes restoration of M2 (to the state that existed at C.5).

IV. Summary

Programmer-transparent coordination of process rollbacks in systems of interacting fault-tolerant processes is a goal pursued in this paper. The approach explored is based on the strategy of prohibiting rollback propagation

due to suspicion, in keeping with the spirit of designing processes to harmoniously cooperate with one another. An important requirement that any practical scheme for automated coordination must meet is the maintenance of a manageable number of process RP's and monitor RP's at all times. Three useful rules were formulated to meet the requirement: maintenance of a minimal set of RP's based on DRP's and PR*'s, safe discard of useful RP's under practical constraints such as limited memory space, and maintenance of a minimal set of monitor RP's.

Implementation of the three rules requires a solution to the problem of communicating validation results among processes. In a system of cooperating processes, it is not practical nor advantageous to make the result of an acceptance test of a process to be immediately known to other processes. It is more practical to make processes to send notices of validation results through (programmer-transparent) "mailboxes" and let processes check their mailboxes at their convenient times. A message communication scheme based on this approach is described in [11].

There is a tradeoff between automated coordination approaches and programmed coordination approaches. Automated coordination involves greater time and space overhead than coordination by the program designer. On the other hand, rollback coordination can often become an unbearable burden on the program designer. Therefore, programmed coordination may become practical only when it is applied at a relatively macroscopic level. In fact, an optimal approach in some situations may be a combination of programmed coordination and automated coordination approaches such that the program designer is concerned only with coordination at a more macroscopic level while coordination at a microscopic level is automatically handled by an intelligent underlying processor system. Such a combination remains as a subject for future study.

Acknowledgment

The author greatly benefited from the discussions with A. Abouelnaga, F. Farrand, J. Huang, and J. H. You during preparation of this paper.

References

[1] A. Abouelnaga, "Validation of recoverable concurrent software systems based on the programmer-transparent coordination scheme," Ph.D. dissertation, Dep. Comput. Sci. and Eng., Univ. South Florida, May 1986.

[2] T. Anderson and R. Kerr, "Recovery blocks in action. A system supporting high reliability," in *Proc. 2nd Int. Conf. Software Engineering*, 1976, pp. 447–457.

[3] P. Brinch Hansen, *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ. Prentice-Hall, 1977.

[4] K. M. Chandy, "A survey of analytic models of rollback and recovery strategies," *Computer*, vol. 8, pp. 40–47, May 1975.

[5] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages*, F. Genuys, Ed. New York: Academic, 1968.

[6] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Inform.*, vol. 1, no. 2, pp. 115–138, 1971.

[7] H. Hecht, "Fault-tolerant software for real-time applications," *Comput. Surveys*, pp. 391–407, Dec. 1976.

[8] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, pp. 549–557, Oct. 1974.

[9] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, *A Program Structure for Error Detection and Recovery* (*Lecture Notes in Computer Science*, vol. 16). New York: Springer-Verlag, 1974, pp. 171–187.

[10] K. H. Kim and C. V. Ramamoorthy, "Structure of an efficient duplex memory for processing fault-tolerant programs," in *Proc. ACM SIGARCH 5th Symp. Computer Architecture*, Apr. 1978, pp. 131–138.

[11] K. H. Kim, "An implementation of a programmer-transparent scheme for coordinating concurrent processes in recovery," in *Proc. COMPSAC'80, IEEE Comput. Soc. 4th Int. Computer Software and Applications Conf.*, Oct. 1980, pp. 615–621.

[12] ——, "Approach to mechanization of the conversation scheme based on monitor," *IEEE Trans. Software Eng.*, vol. SE-8, no. 3, pp. 189–197, May 1982.

[13] ——, "Software fault tolerance," in *Handbook of Software Engineering*, C. R. Vick and C. V. Ramamoorthy, Eds. New York, Van Nostrand Reinhold, 1984.

[14] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220–232, June 1975.

[15] C. S. Repton, "Reliability assurance for System 250: A reliable, real-time control system," in *Proc. Int. Computer Communication Conf.*, 1972, pp. 297–305.

[16] J. A. Rohr, "STAREX-self-repair routines: Software recovery in the JPL-STAR computer," in *Dig. 1973 Int. Symp. Fault-Tolerant Computing*, pp. 11–16.

[17] D. L. Russell, "State restoration in systems of communicating processes," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 183–194, Mar. 1980.

[18] S. K. Shrivastava, and J. P. Banatre, "Reliable resouce allocation between unreliable processes," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 230–241, 1978.

[19] S. K. Shrivastava, Ed., *Reliable Computer Systems*. New York, Springer-Verlag, 1985.

K. H. (Kane) Kim (S'73-M'75-SM'86) received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1969, the M.A. degree in computer science from the University of Texas, Austin, in 1972, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1974.

From 1969 to 1971 he served as an officer in the Korean Army. From 1975 to 1986 he served on the faculty of the University of South Florida, Tampa, the State University of New York, Binghamton, and the University of Southern California, Los Angeles. While teaching at Binghamton, he also served as acting chairman of the Department of Computer Science for nine months. He is currently a Professor of Computer Engineering in the Department of Electrical Engineering at the University of California, Irvine. His current research interests are in the areas of reliable distributed processing and real time software engineering. He is currently conducting both analytic and experimental research in the DREAM (Distributed Real Time Ever Available Microcomputing) Laboratory.

Dr. Kim is a member of the Association for Computing Machinery and the IFIP Working Group 10.4, and a senior member of the IEEE Computer Society. He served as the Chairman of the IEEE Computer Society's Technical Committee on Distributed Processing from September 1984 to December 1986. In 1989 he plans to host the IEEE Computer Society's 9th International Conference on Distributed Computing Systems as the General Chairman.

Appendix A.IX

Issues in Design of Temporary Blackout Handling Capabilities into Tightly Coupled Computer Networks

Issues in Design of Temporary Blackout Handling Capabilities into Tightly Coupled Computer Networks

K.H. (Kane) Kim
Computer Engineering Program
Dept. of Electrical Engineering
University of California
Irvine, Calif. 92717

Abstract

The temporary blackout (TB) that disrupts orderly operation of electronic components and erases the contents of registers and RAMs occurs in many applications due to unreliable power sources, high energy events, etc.

In order to identify the design issues in providing TB handling capabilities in distributed computer systems, an experimental design of the TB handling capability into a real-time tightly coupled network testbed has been conducted.

This paper discusses what was learned through this experimental study, especially, various design issues that are encountered in designing TB handling capabilities into tightly coupled computer networks. Promising design approaches including those for structuring state-saving actions and cooperative recovery actions of distributed processes are also discussed.

1. Introduction

Many challenging real-time applications, e.g., space exploration applications, require computer systems to be capable of surviving through temporary blackout (TB) events that disrupt orderly operation of electronic components and erase the contents of registers and RAMs. Such a TB may be caused by unreliable power sources or high energy events. While TB handling in a uniprocess system is a problem studied for quite some time, designing TB handling capabilities into tightly coupled computer networks (TCN's) which are now demanded in many applications, is a challenge to the system designer.

The purpose of this paper is to delineate various design issues that are encountered in designing TB handling capabilities into TCN's. The paper starts in Section 2 with the discussion on the criticality of TB handling in some applications. Section 3 then deals with the basic tools needed and operations involved in handling TB's. The discussion here focuses on the relatively simple case of the uniprocess systems. The issues in designing TB handling capabilities into TCN's are

discussed in Section 4. Promising design approaches including those for structuring state-saving actions and cooperative recovery actions of distributed processes are also discussed. As a way of identifying specific design issues and promising approaches, an experimental design of the TB handling capability into a real-time tightly coupled network testbed has been conducted at the author's former and current affiliations, the University of South Florida (USF) and the University of California, Irvine (UCI). Observations made during this experiment conducted as a case study are briefly discussed in Section 5. Section 6 is a summary.

2. Criticality of Temporary Blackout (TB) Handling

The importance of providing TB handling capabilities in computer systems for unattended long-mission applications such as many space-borne applications has long been accepted by system designers. When the probability of a TB event occurring during the mission life in such an application is reflected, one can quickly come to the conclusion that a system not equipped with TB handling mechanisms can in no way meet the reliability goal set for the application. Therefore, even if incorporation of TB handling mechanisms means to multiply the system cost several times, the system designers have no choice.

On the other hand, the short mission-life applications in which the probabilities of TB events occurring during the mission lives are non-negligible have been dealt with relatively infrequently. More such applications are now being explored. Suppose that a given application has the mission life of 10 minutes. Suppose also that the reliability goal in this application is to have a computer system capable of surviving through the 10-minute mission with the probability of $(1-10^{-3})$ $(=0.999)$. If the probability of a TB occurring during the mission is 10^{-3} $(=0.001)$, it is impossible to meet the system reliability goal without use of a TB handling scheme. Let us now define the coverage of the TB handling scheme as the probability of successfully containing the effects of TB's occurring during the mission and preventing critical damages to the application computation. If the coverage of the TB handling scheme is only 0.5, then the reliability attainable by a system with this TB handling scheme cannot exceed $(1-5 \times 10^{-4})$ $(=0.9995)$.

Therefore, even in some short-mission applications, TB handling is an essential requirement.

3. Temporary Blackout (TB) Handling in Uniprocess Systems

In this section, basic tools used and operations involved in handling TB's in uniprocess sytems are discussed.

3.1 Hardware requirements and basic operations

The essence of TB handling is to periodically establish recovery points by saving the critical state variables into hardened non-volatile store and after experiencing a TB, conduct forward recovery by use of the saved state variables and the time information.

Therefore, the two basic operations involved are state saving and recovery. A hardened non-volatile storage device capable of keeping its contents intact through TB periods is thus an essential requirement. Such a device is called a safe storage device in this paper. The safe store may consist of read-only storage devices and read-write devices. Read-only devices can contain the code of both the operating system and the application program whereas read-write devices are needed to keep values of critical state variables.

Another hardware requirement is in the area of detecting the beginning and the termination of a TB. An arrangement must be made for the system to attempt a safe shutdown to the extent possible upon arrival of a TB initiating condition. Similary, the system must be designed to wake up and initiate a recovery as soon as possible upon termination of a TB condition.

A hardware device essential in facilitating a recovery after a TB is a hardened real-time (calendar) clock. A measurement of the duration of the TB that has just expired is needed in forward recovery. The forward recovery procedure is application-dependent in general. However, it basically involves first restoring critical state variables with the most recently saved values, next reading the current status of the application environment, and finally establishing the computation to an appropriate state compatible with the current environment. A hardened clock may be provided internally within the system or an external hardened clock may be utilized.

3.2 State saving

As mentioned above, one of the two basic operations involved in TB handling is the state saving. Major issues in design and implementation of the state saving operation are summarized below.

The set of critical state variables that are saved for the purpose of TB handling is called the State vector in this paper. Important design parameters here are when and how often to save the state vector and how to choose the membership of the stae vector. A key factor that impacts these design decisions is whether some of the sensor data received or the actuator commands prepared to send out may be lost (due to TB's) without endangering the application.

In determining the membership of the state vector, an extreme approach aimed for minimizing the loss of sensor data or actuator commands is to save all the variables. This approach can also relieve the system designer of the burden of selecting variables as members of the state vector. However, it has a drawback of making the size of the state vector large which in turn makes the saving time and the restoration time large. This approach is not usable in some

applications because of the large saving time it incurrs. An explicit determination of the state vector is inevitable in such applications.

One way of reducing the time overhead incurred by the state saving operation is to exploit parallelism. The basic idea is to use an auxiliary processor dedicated to saving the state vector as well as a mechanism for storing the same value into multiple locations simultaneously. Several proposals for such an auxiliary processor capable of saving the state vector concurrently with the application processing by the main processor have appeared in the literature, but none of them have yet been prototyped to this author's knowledge.

A point in execution when the state vector is saved is called a saving point, or recovery point in this paper. Location of saving points is another important design parameter. There are basically two choices. One is to establish saving points independent of the application program structure, e.g., clock-driven location. This approach can work with the approach of including all variables into the state vector but does not match well with the selectively determined state vector. The other approach is to have the system designer place saving instructions in strategic locations within the application program structure. The placement decision may be influenced by the membership of the state vector selected as well as by the recovery time limit.

3.3 Recovery

As mentioned in Section 3.1, the post-TB recovery typically involves 1) restoration of the state vector, 2) recognition of the environment state, and 3) rapid adjustment of the computation to a state compatible with the environment. This forward recovery procedure is necessary because the pure rollback-and-retry is not feasible due to the time constraint as well as the inability to capture the information that might be available during the TB period. In fact, if the TB duration exceeds a cetain threshold which differs among applications, the recovery becomes impossible. Understanding of the threshold is important in analyzing the expected reliability of an application system that must endure TB's. If some hardware resources are damaged during a TB, then the hardware diagnosis and reconfiguration should be performed as a part of the post-TB recovery.

4. Temporary Blackout (TB) Handling in Tightly Coupled Computer Networks (TCN's)

Compared to the case of a uniprocess system, TB handling in a TCN has an added dimension of complexity. In both state saving and recovery, distributed cooperating processes are involved and a number of new design issues arise. Some of these issues are discussed below.

4.1 State Saving

First of all, distributed processes must establish their recovery points (i.e., saving points) in a coordinated manner; otherwise, the TCN that has experienced a TB may not be able to restore itself to a consistent state from which an attempt can be made for forward recovery. Uncoordinated recovery points may also lead to an avalanche of rollbacks [Ran75]. A coordinated set of recovery points of interacting processes that define a consistent system state is called a recovery line (or saving line). A safe approach to establishment of recovery lines in TCN's is to have the sytem designer to carefully place saving instructions in each process engaged in distributed computation as adopted in the conversation scheme [Ran75, Kim82]. Although this approach imposes substantial burdens on the designer, it is the most appealing in applications subject to severe time constraints since very little run-time decisions are involved in state saving. In order to minimize the amount of computation lost due to TB's, it is desirable to design saving activities so that the probability of the distributed saving activities occurring simultaneously may be maximized. A high-resolution global clock can be useful here. Approaches that impose less burdens on the designer and require less coordination of process designs but more coordination-oriented process actions at run time have been discussed in the literature but their applicability is limited to TCN's in applications subject to soft time constraints. Nevertheless, this is an area requiring further study.

The structure of the safe store is another design parameter. The distributed store structure that supports concurrent saving activities of distributed processes is desired but may be too costly in some applications. If access conflicts among distributed processes during their state saving operations are possible, then the problem of designing the saving activities to yield optimal performance becomes further complicated.

4.2 Recovery

The state vector saved at a recovery line in a TCN can be viewed as a network state vector consisting of process state vectors, each belonging to a different process engaged in distributed computation. Upon termination of a TB, each process restores its state vector with the most recently saved values and attempts to adjust itself to a state compatible with the new environment and be ready for normal processing. Here different processes may take different amounts of time to become ready for normal processing. If processes are allowed to proceed as soon as they become ready, i.e., to make an asynchronous restart, a provision must be made to prevent an excessive gap developing among processes in their restart times; otherwise, a "false alarm" by an early process that announces the death of another process which is actually alive may occur. A simple approach here may be to make the recovered processes wait until all the cooperating processes have recovered before making a synchronous restart. However, it may be advantageous in many situations to allow processes to make an asynchronous restart. Designing distributed processes capable of performing forward recovery followed by asynchronous restart is not a well understood subject at present.

5. An Experimental Design of a TB Handling Scheme

In order to identify specific design and implementation issues and promising approaches, an experimental design of a TB handling scheme into a real-time TCN testbed has been conducted at the author's former and current affiliations. The testbed used is called the on-board intelligence (OBI) testbed and is based on a computer network called the Crossbar Multi-microcomputer System (CMS) produced by Unisys Corp. in Huntsville, AL for the US Army [Chu87,McD82]. The CMS is composed of six microcomputers and a program-loading host computer interconnected to a maximum of 12 global shared memory modules through a crossbar switch. A distributed operating system kernel and a real-time distributed processing application software were developed to run on the CMS. The application implemented is depicted in Figure 1 and it includes simulators of various sensor devices and actuators used in space vehicles.

Three nodes (Nodes 2, 3, 4) simulate three different parts of the application environment together with three different types of sensor devices and controllers. In addition, each of the three nodes performs some data processing functions, more specifically, some preprocessing of the sensor data and some postprocessing of the abstract control commands coming from the main control processor (Node 1). Node 4 is a little different from other nodes in that it provides sensor data to the main control processor but does not need control commands from the main processor. It generates control commands by itself.

TB handling mechanisms were then incorporated into the testbed. Subsequently, simulated TB's were injected and performance indicators such as recovery time were measured. The study here is primarily focused on the following questions.

(1) How difficult is it to implement an effective forward recovery logic?
(2) How complex is the logic of TB handling?
(3) How fast can the TCN recover after experiencing a TB?
(4) How long a TB can the TCN endure without failing to meet the application requirements?

TB handling in the OBI testbed environment has turned out to be simpler than in many other computer network application environments. There are two factors in this testbed that makes TB handling relatively simple.

(1) The main control processor plays the role of a coordinator with respect to the overall application. Moreover, it was assumed that no procesor would be completely lost during the experiment since the objective was to focus on the study of TB handling. Therefore, asynchronous restart of recovered processes was easy to implement because every recovered process was designed to merely check the health of its local sensor and controller until the main control process came alive.

(2) The message traffic among the nodes was relatively low and at a reasonably steady rate. The CMS is equipped with a high-resolution global clock. These plus the coordinator - follower relationship

existing between the main control processor and other nodes made it relatively simple to implement periodic establishment of recovery lines by use of the global clock.

Although the experiment is still continuing, some useful insights and data have already been obtained. In the OBI testbed, the forward recovery logic was not very complicated and proved to be effective. It is expected, however, that the complexity of the logic will increase rapidly as additional application requirements are introduced and more distributed processes are added to the testbed. The amount of critical variables saved by each node was about 100 bytes on the average. Therefore, the state saving overhead is modest in view of the duration of the control cycle in this application being in the order of tens of milliseconds. The data obtained also indicated that in the case of the chosen application, recovery could be accomplished in less than 0.5 milli-seconds when implemented with the off-the-shelf components.

6. Summary

Various design issues encountered in designing TB handling capabilities into real-time uniprocess systems and TCN's have been discussed. The set of issues discussed has been identified through limited experiences and can in no way be regarded as comprehensive. It is fair to say that TB handling in distributed computer systems is a young technological field. Much more research, especially testbed-based experimental research, is needed.

Acknowledgement: The work reported here was supported in part by the U.S. Army, SDC and the NASA JPL under Contract No. NAS7-918-RE-182/443, and in part by the Office of Naval Research under Contract No. N00014-87-K-0231.

References

[Chu87] Chu, W.W., Kim, H.H., and McDonald, W.C., "Testbed-based Evaluation of Design Techniques for Fault-Tolerant Real-Time Distributed Computer Systems," Proceedings of the IEEE, Vol.75, No.5, Special Issue on Distributed Databases, May 1987, pp.649-667.

[Kim82] Kim, K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitor," IEEE Trans. on Software Eng., Vol. SE-8, No.3, May 1982, pp.189-197.

[McD82] McDonald, W.C., and Smith, R. Wayne, "A Flexible Distributed Testbed for Real Time Applications," IEEE Computer, Vol.15, No.10, Oct. 1982, pp.25-39.

[Ran75] Randell, B., "System structure for software fault tolerance," IEEE Trans. on Software Engr., June 1975, pp.220-232.

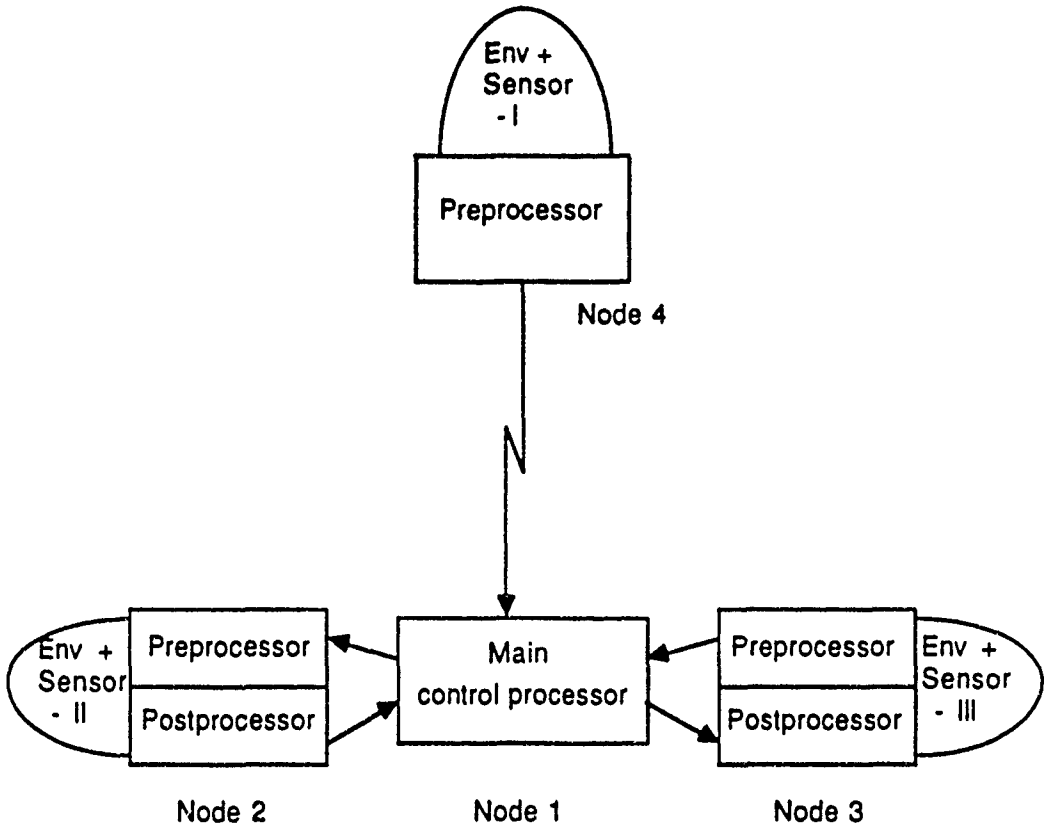


Figure 1. A real-time application system implemented on the CMS

Appendix A.X

An Approach to Experimental Evaluation of Real-Time Fault-Tolerant Distributed Computing Schemes

An Approach to Experimental Evaluation of Real-Time Fault-Tolerant Distributed Computing Schemes

K. H. (KANE) KIM, FELLOW, IEEE

Abstract—A testbed-based approach to the evaluation of fault-tolerant distributed computing schemes is discussed. The approach is based on experimental incorporation of system structuring and design techniques into real-time distributed computing testbeds centered around tightly coupled microcomputer networks. The effectiveness of this approach has been confirmed through some experiments conducted in the author's laboratory. Primary advantages of the testbed-based approach include the relatively high accuracy of the data obtained on timing and logical complexity as well as the relatively high degree of assurance that can be obtained on the practical effectiveness of the scheme evaluated. This paper discusses various design issues encountered in the course of establishing the basic microcomputer network testbed facilities and augmenting them to support some experiments conducted. The shortcomings of the testbeds that have been recognized are also discussed together with the desired extensions of the testbeds. Some of the desired extensions are beyond the state of the art in microcomputer network implementation.

Index Terms—Distributed computer system, distributed recovery block, experimental validation, fault tolerance, real-time computer system, recovery, temporary blackout, testbed.

I. INTRODUCTION

As the complexity of distributed computer systems (DCS's), especially those in critical applications, continues to grow, there are increasing calls for rigorous validation of the system structuring techniques and operating schemes [16], [10], [14]. Since analytic approaches still fall short of handling complex systems with sufficient accuracy, experimental validation techniques have become highly desired. Although the field of experimental validation of complex DCS's is still in its early stage, recent drastic reduction in the costs of building-blocks of DCS's, especially microcomputers and connection devices, made prototyping and experimentation an effort that is within the domain of numerous research groups.

A few testbeds have been established to support study of issues in real-time system design [6], [7], [16]. For example, the MARS project at the Technical University of Vienna has been using a local area network of MC68000-based nodes equipped with VLSI clock synchronization chips to establish a real-time DCS testbed [14], [15]. The Cronus project at BBN, Inc., Massachusetts, and the HOPS project at Honeywell, Inc., Minnesota, have been using workstation networks with UNIX as the underlying operating system to create testbeds for object-oriented distributed real-time systems [4], [6], [19]. Many other testbeds established or under construction are aimed at supporting study of issues in fault-tolerant distributed database design. For example, the RAID testbed at Purdue University and the CARAT testbed at the University of Massachusetts are distributed transaction processing system testbeds built around networks of workstations and super minicomputers [1], [6], [13]. Numerous examples of testbed efforts related to some aspects of distributed computing can be found in [6].

At the author's current institution, University of California, Irvine (UCI), and his former affiliation, University of South Florida (USF), attempts have been made over the past several years to establish low-cost testbed facilities for validation of various fault-tolerant distributed computing techniques. In order to keep the cost of the facilities low, microcomputers and low-cost connection devices such as serial broadcasting bus (e.g., Ethernet [20]) and RS232C serial point-to-point connections, have been used as primary building-blocks. At present the testbed facilities established represent the core of a laboratory named the Distributed Real-time Ever Available Microcomputing (DREAM) Laboratory at UCI. Major characteristics of the experimentation efforts conducted in the DREAM Laboratory are as follows.

1) The approaches adopted for validation of design and execution techniques are based on experimental application of the techniques to the microcomputer network testbeds and measurement and evaluation of various aspects of the resulting microcomputer networks such as fault detection and recovery performance, expandability, etc.

2) The research efforts have been focused on real-time distributed computing applications.

3) The design techniques under study are primarily those intended for achieving *system level fault tolerance*, i.e., tolerance of both hardware and software faults in DCS's [8], [11], [18] Development of efficient implementation techniques for such system-level fault tolerance

Manuscript received March 1, 1987; revised March 31, 1988. This work was supported in part by the NASA JPL and the U.S. Army SDC under Contract NAS7-918-RE-182/443, in part by the Office of Naval Research under Contract N00014 87 K-0231, and in part by the University of California MICRO Program and AT&T under Grant 88 123.

The author is with the Computer Engineering Program, Department of Electrical Engineering, University of California, Irvine, CA 92717.

IEEE Log Number 8927 83.

schemes is also a major objective of the experimental research being carried out in the laboratory.

The need for experimental validation is particularly acute in the case of fault-tolerant DCS's. A large number of proposals have appeared in literature for achieving fault tolerance in DCS's. Yet very few have been convincingly demonstrated. Testbed-based validation efforts are most desired. However, such efforts are feasible only when economic, high performance, and user-friendly testbeds are available. Although the hardware situation has now sufficiently improved to allow such testbed establishment activities, the software situation is still far short of the desired state.

This paper is a discussion on various design issues encountered in the course of establishing the basic microcomputer network testbed facilities and augmenting them to support some fault-tolerant distributed computing experiments conducted. Although both tightly coupled network (TCN) and loosely coupled network (LCN) testbeds have been established and both the fault tolerance schemes suitable for TCN applications and those for LCN applications have been experimented with, only those related to TCN testbeds will be dealt with in this paper. Section II deals with basic TCN hardware configurations while TCN software facilities are discussed in Section III. The hardware facilities are centered around two TCN's, one called the Macro-Dataflow Network (MDN) and the other the Crossbar Multi-microcomputer System (CMS). The software facilities include internally produced real-time distributed operating systems, language tools, and real-time application software. Section IV describes two fault tolerance schemes that have been incorporated into TCN testbeds. It also discusses the instrumentation made of the testbeds to support the experiments and briefly touches upon the measurement results. The shortcomings of the testbed facilities that showed up during the experimentation are discussed in Section V together with the desired extensions of the testbeds. The final section is a summary section.

II. TCN Hardware Configurations

The TCN hardware facilities are centered around two major pieces of equipment: the internally produced Macro-Dataflow Network (MDN), and the Crossbar Multi-microcomputer System (CMS) produced by Unisys Corporation in Huntsville, AL, for the U.S. Army [17]. Additional hardware equipment includes graphic status-display tools, and various microcomputers equipped with disks serving as software development workstations.

A. Macro-Dataflow Network (MDN)

In real time systems, a typical control cycle starts with the generation of data by sensor devices and ends with command output delivered to certain controller devices. Between the two events, i.e., sensor data input and control command output, there are various data processing steps involved. In the MDN, the data processing steps are distributed among different microcomputer nodes for the sake of throughput increase. In a sense, sensor data travel through various *computing stations* (microcomputers executing processing steps) before they reach the output station. During the traveling, the data may get transformed or replaced by drastically different types of data, of course. In this paper a *data set* is defined as a set of data that is communicated between computing stations and activates an execution of a processing algorithm (step). Typically, queueing of data sets takes place between computing stations in order to maximize concurrency of their operations.

In order to achieve fast data turnaround in the MDN, it is essential that internode communication be highly efficient. Otherwise, control cycles may fail to complete within deadlines and even the total system throughput may become worse than that of a centralized processing system. The internode connection approach adopted in the MDN is based on the use of a two-port buffer memory as a medium for connecting a pair of microcomputers. The microcomputers are Z8001-based single-board microcomputers named OEM-Z8000's. They currently operate at the rate of 4 MHz.

The amount of on-board RAM on an OEM-Z8000 currently ranges from 32 to 120 Kbytes. The access time of the two-port buffer memory developed in house is the same as that of the on-board memory of the OEM-Z8000. A sufficient number of these buffer memory modules were constructed to configure a variety of network topologies involving six microcomputer nodes. Each dual-port memory module consists of two separately accessible memory banks as shown in Fig. 1. Therefore, while one OEM-Z8000 is accessing a memory bank through one port, another OEM-Z8000 can access the other bank in the same buffer module through the other port. This also means that by alternatively using the two banks as communication buffer between two OEM-Z8000's, the time that an OEM-Z8000 has to spend waiting for a buffer being accessed by the other OEM-Z8000 to be released can be significantly reduced.

Fig. 2 shows an example of the MDN configurations that have been established. The example configuration consists of six nodes linked with 10 dual-port buffer memories. Each node in the network is housed on a wall backplane card and consists of an OEM-Z8000 microcomputer and a local memory extension board. Each dual port buffer memory board is housed on the backplane card that holds one of the two OEM-Z8000's connected to the buffer memory. Since flat ribbon cables are used as media for connecting OEM Z8000's to the ports of the buffer memory boards, change of the network topology is not difficult. The OEM-Z8000 is equipped with an interval timer and two serial I/O ports. The timer generates 1200 interrupts per second. It is used to construct a software implemented real time clock in the MDN.

One advantage of this MDN architecture for use as a core of a fault tolerance testbed is that it has a minimal set of components shared among processing nodes. This is particularly true compared to a single common serial

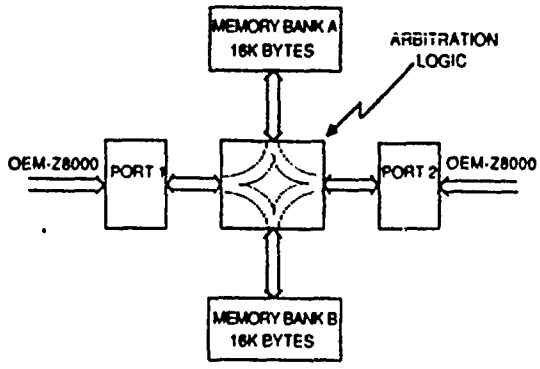


Fig. 1. The structure of the two-port buffer memory.

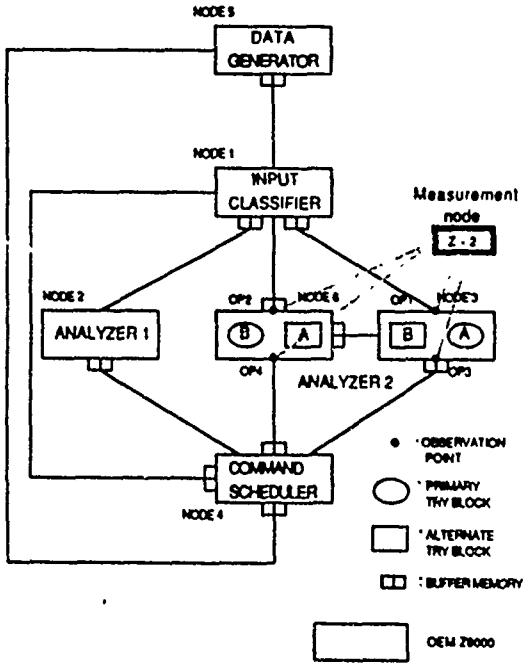


Fig. 2. A configuration of the macro-dataflow network (MDN).

bus based network such as Ethernet or a global memory based network where multiple processing nodes use a global memory module as medium for interaction. Minimization of shared components is desirable not only with respect to concurrent processing but also with respect to fault tolerance. The fewer shared components the network contains, the easier it is to prevent undesirable interference among processing nodes; it is then easier to assess and contain the damages caused by faults. In some sense, the minimization of shared components stems from the intrinsic nature of the MDN architecture, that is, the network structure of the MDN closely matches the computation structure of a chosen real-time application. There are obvious disadvantages, though, such as limited run-time reconfigurability and costs associated with the relatively large number of connections used.

B. Crossbar Multimicrocomputer System (CMS)

The design philosophy underlying the CMS is different from that for the MDN in that regular structure and general-purpose nature of the architecture were important driving factors in determination of the architecture of the CMS. It was designed both for flexibility needed in studying a range of fundamental architectures and for maximum interconnectivity.

The CMS is composed of six microcomputers and a program-loading host computer interconnected to a maximum of 12 global shared memory modules through a crossbar switch as illustrated in Fig. 3. Microcomputer nodes in the CMS are also OEM-Z8000's. Each global memory module is accessed by normal Z8001 memory read/write operations. Therefore, instructions may be executed from either global or local memory. Each global memory module contains an arbitration unit that implements a first-come-first-serve policy for resolution of conflicting memory requests. In the absence of contention an OEM-Z8000 can access a location in a global memory module in the same amount of time that is required to access a local RAM.

The CMS also provides special functions such as a global real-time clock, inter-microcomputer interrupt facilities, and synchronization flag memory, all in the form of memory-mapped peripheral facilities. The real-time clock provides a common 31 bit, 500 KHz time source accessible by all microcomputers concurrently for performance evaluation. In addition to this global clock, each OEM-Z8000 contains an interrupting interval timer that can be used for future event scheduling or for setting watchdog timers. Inter-microcomputer interrupt facilities enable any microcomputer to interrupt any other microcomputer, or the host machine. The flag memory provides 16,384 single-bit flags that can be used for synchronizing access to global memory. Each flag is assigned two addresses and a read request to a flag is interpreted as a "test-and-set" request with the flag set to "0" or "1" depending upon the address used.

The CMS is a quite flexible tool for simulating a variety of TCN architectures. The flexibility stems from the full connectivity provided between microcomputers and global memory modules. It is also due to the potential of a global memory module for use in simulating a point-to-point communication link or a broadcasting channel to some degree of accuracy.

III. TCN Software Facilities

A. MDN Software Structure

Two major decisions were made at the early stage of MDN software development. One was to structure the real-time application software to run on the MDN as cooperating processes. Since much of the operating system functions was to be implemented as cooperating processes as in many other operating systems, the decision meant that there would be two different types of processes supported by the kernel: application processes and system processes. The primary motivation here was to minimize software layering so that execution overhead may become more affordable in real-time applications.

The other decision was to design all application software and operating system processes with the aid of high-

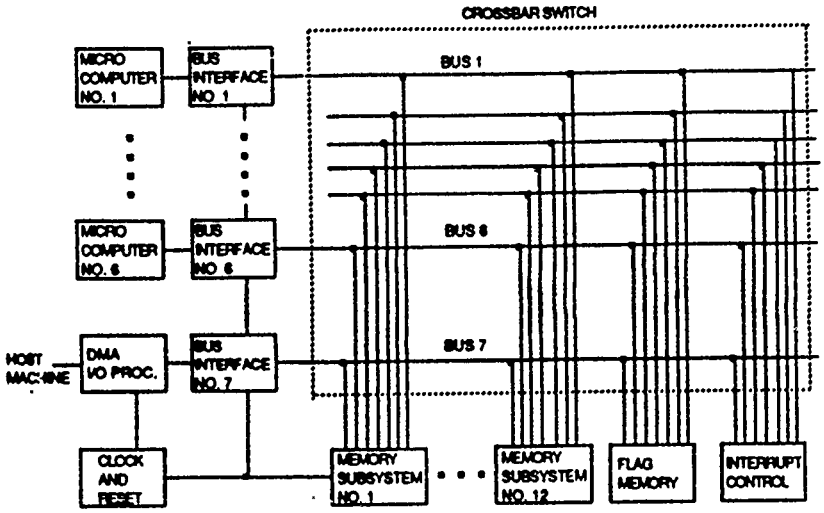
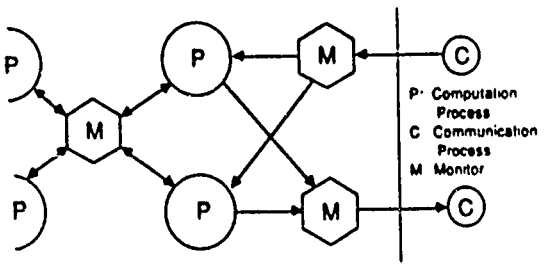


Fig. 3. Crossbar multimicroprocessor system (CMS) (adapted from [16]).

level concurrent programming languages. This was of course motivated by the desire to achieve better program portability and debugging efficiency. Processes produced with such aid require a software kernel that supports their concurrent execution and handles low-level I/O with peripheral devices. In a sense, such a software kernel and the machine hardware together form a *virtual machine* executing concurrent programs. A virtual machine called the Extended Concurrent Pascal Machine (ECPM) was implemented around an OEM-Z8000. The ECPM structure and other major software components are discussed in this section.

1) ECPM: The name ECPM was adopted because the ECPM was designed to support cooperating processes designed in an extension of programming language Concurrent Pascal [2], [10]. The driving philosophy in designing the ECPM was to pursue simplicity as far as possible. The simplicity was considered to be an attribute of great importance if the timing behavior of the system were to be clearly understood. Fig. 4 depicts the services provided by the ECPM. Besides the typical basic kernel functions such as system initialization, I/O with peripherals, interrupt handling, locking, and process context switching, three additional essential functions were incorporated into the ECPM. They are networking, time management, and flexible process scheduling.

As mentioned in Section II, a dual-port buffer memory is used in the MDN as the basic hardware link that facilitates networking. It consists of two independent memory banks. In order to use a memory bank, an OEM-Z8000 must request for the right to access to the bank. When a request is granted, the OEM-Z8000 may read and change the contents of the memory bank at the same speed at which it works with its local memory. When finished, the memory bank must be released so that the other OEM-Z8000 may access it. This buffer memory access protocol together with simple nonblocking message send/receive primitives form the core of the networking service that the ECPM provides. The simple message passing primitives were accepted to ensure deterministic timing behavior of



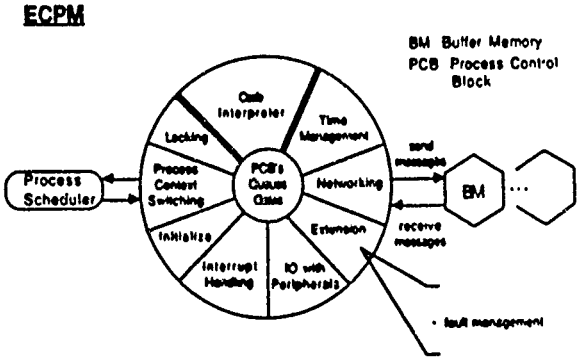


Fig. 4. The MDN software structure.

the networking subsystem. The exact virtual machine instructions that are supported include the following: 1) "obtain access right (buffer memory bank)," 2) "read message (local memory, buffer memory)," 3) "write message (local memory, buffer memory)," and 4) "release access right (buffer memory bank)."

The time management services provided by the ECPM are also simple. They include time slicing, watchdog timer control, fixed period waiting and continuous real-time clock maintenance. Time slicing was provided to support study of various process scheduling strategies The continuous clock is implemented in software with an interval timer.

Conceptually, it seems more logical to place the function of scheduling ready processes for execution above

the ECPM and make it a part of a system process while only the function of dispatching the next process to run is left to the ECPM. However, we learned through experimentation that this approach resulted in poor system performance. On the other hand, it was clear that process scheduling in real-time systems is not a well-understood subject and we would have to study and experiment with various scheduling strategies for a long period of time. Therefore, the decision accepted was that the scheduling module be located within the ECPM, but made a separate module that was accessed from the rest of the ECPM through a well-defined interface and could be modified and reassembled without involving the rest of the ECPM. Preparation of scheduling parameters remained a responsibility of a system process running above the ECPM, though. This means that the ECPM supports a virtual machine instruction "set-scheduler (parameters)." In concurrent software systems designed in Concurrent Pascal, there is a special system process called the initial process which is the only process capable of creating other processes. The "set-scheduler" instruction is usually used by the initial process.

Adapting a new scheduling strategy might require replacement of the existing scheduling module with an entirely new module. One scheduling strategy that we have adopted for experimental study because of its flexibility and efficiency is a dynamic priority scheduling strategy called the "priority bracket" scheduling strategy [9]. Each process is assigned a priority bracket representing an upper bound and lower bound on the priority that the process may assume during its life-time. The programmer sets a priority bracket for each process by designing the initial process to pass the priority bracket information to the scheduler. Simple strategies such as the round-robin strategy and the fixed priority strategy that have been frequently used in the systems designed so far are special cases of the priority bracket scheduling strategy. Further details are referred to [9].

Discussion of the extensions made to the ECPM to support fault tolerance experiments is deferred to Section IV. Fig. 4 also shows a code interpreter as a part of the ECPM. This is in a sense an interface through which processes request for ECPM services.

2) System Processes: In the basic MDN configuration, system processes provide two basic functions: initialization of the system and message communication with other nodes.

System initialization includes setup of cooperating process configurations and the scheduling parameters for the processes, and these are usually done by the initial process. Intermode message communication was initially handled by specialized system processes. These specialized processes were called *communication processes* whereas other application oriented processes were called *computation processes* Fig 4 illustrates the relationship between computation processes and communication processes Basically the latter provides messenger service to the former Communication processes can be classified into two types, input communication processes and output communication processes.

A computation process wishing to transmit a message to another residing in a different node simply deposits the message into a buffer monitor [2] connected to a communication process. The communication process then deposits the message into a dual-port buffer memory which will be checked by a communication process at the receiving node. The latter process deposits the incoming message into a buffer monitor connected to the destination computation process.

Although the above approach has the advantage of producing a modular structure and allowing the location-independence of application processes by isolating location-dependent functions within the communication processes, the actual implementation showed poor network performance. The reason was mainly because the buffer memory path was so efficient that the extra delay in a message path due to the presence of communication processes represented significant increase in the total transmission time from one computation process to another remote process. It should also be noted that the MDN is not equipped with disk storage, which dictates keeping of all files, at least the files needed during real-time processing, in main memory. Such main memory resident files are not unusual in real-time applications. We felt that the increased internode communication time could not be justified in the time-critical applications where MDN-type of systems are needed. Therefore, communication processes were eliminated and computation processes were made to directly access the buffer memory.

In order to support fault tolerance experiments, system processes had to be extended with additional functions. Such functions will be discussed in Section IV.

3) Application Processes: In order to support experimental study of various TCN system design techniques, a real-time application was implemented on the MDN together with a simulator of the application environment. Fig. 2 depicts the MDN configuration running the application program. Node 5, also called the data generator node, simulates the application environment together with sensor and controller devices. In general, once the sensor devices produce data, then the DCS occupying the rest of the MDN must respond with control output within certain deadlines. If these deadlines are violated, the environment may run into an undesired state or the sensor devices may not be able to produce useful data from there on. The data generator node is driven by the real-time clock and its rate of progress cannot be influenced by other nodes of the MDN. It is a continuous nonterminating simulator.

The remaining five data processing nodes of the MDN execute the input classification step (node 1), various analysis steps constituting the intelligence of the solution algorithm (node 2, node 6, node 3), and a control command scheduling step (node 4) that delivers the network's response to the controller device. The stimulus data from node 5 are first handled by the input classification node which distributes inputs to the rest of the network. The

command scheduler honors requests from various analysis steps to schedule commands for the controller device.

Each analysis step, i.e., the Analysis-1 step executed on node 2 and the Analysis-2 step executed on node 6 and node 3, consists of several loosely related substeps and was initially designed as a set of a few cooperating processes. If there were enough microcomputer nodes, each process could have been assigned to run on a dedicated node. However, multiplexing a set of cooperating processes on each node resulted in poor performance primarily due to the scheduling overhead. Subsequently, the cooperating processes on each node have been merged into a single process. The analysis process that resulted executes one of several tasks depending upon the input data set it receives. Considering the downward trend in both the cost and the physical volume of microcomputers, dedicating a microcomputer node to run a single computation process is regarded as the proper direction to follow.

If the rate of data generation goes up, then additional analyzer nodes, both of Analyzer-1 and Analyzer-2 types, may be added for higher degree of multiprocessing. Use of up to several tens of analyzer nodes is envisioned in the chosen application. (Since early 1985, the non-fault-tolerant version of this real-time application program has been running reliably on the MDN.)

4) Programming Language Tools: Concurrent Pascal with our extensions related to process scheduling and communication between remote processes, has been the primary language tool for designing cooperating processes. The main advantages of this choice are the simplicity and the abstract data type oriented style of the language. The language and its compiler were simple enough for us to extend the language without much difficulty as we encountered new requirements. The main disadvantages are the lack of flexible interprocess communication mechanisms, the lack of modular compilation capability, and the slow interpretive execution approach used. Overall, it has been a cost-effective rapid prototyping tool for development of MDN software.

In order to balance the workload in the MDN, certain nodes had to be sped up significantly after the initial operational version of the application program had been obtained. In particular, the data generator node became a bottleneck and thus its application process was rewritten in programming language C. (Other tools such as PDL developed by Unisys Corp., Modula-2, and Ada have been obtained but not yet used as research tools.)

B. CMS Software Structure

1) Kernel: The CMS kernel developed internally is simpler than the ECPM in the MDN. (The ECPM can also be made to fit into the CMS with minor modification, but it has been left as a future task.) The simple CMS kernel has been designed to support processes written in programming language C, each running on a separate OEM-Z8000 node. Besides the typical service functions such as system initialization, I/O with serial communication ports, locking, and interrupt handling, the kernel provides additional services. They are synchronized restart, global clock reading, and message communication through global shared memory.

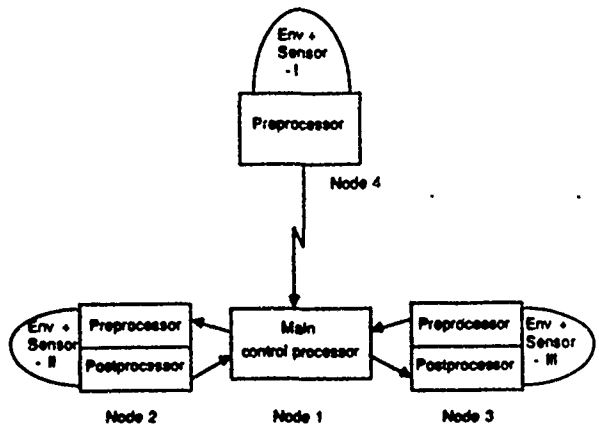


Fig. 5. A real-time application system implemented on the CMS.

The processes wishing to make a synchronized restart will wait inside their supporting kernels for a start signal to be generated by a designated coordinator node in the CMS. In order to support message communication between a pair of distributed processes, the kernel implements a circular message queue with a lock on a selected global memory module.

2) Application Processes: As in the case of the MDN, a real-time application was implemented on the CMS together with a simulator of the application environment. Fig. 5 depicts the distributed application software configuration. Unlike the MDN application software depicted in Fig. 2, the CMS application implemented is a multisensor control application. As shown in Fig. 5, three nodes (Node 2, 3, 4) simulate three different parts of the application environment together with three different types of sensor and controller devices. In addition, each of the three nodes performs some data processing functions, more specifically, some preprocessing of the sensor data and some postprocessing of the abstract control commands coming from the main control processor (Node 1). Node 4 is a little different from other nodes in that it provides sensor data to the main control processor but does not need control commands from the main processor. It generates control commands by itself.

As mentioned in the preceding section, each node runs one process only. There is no separate system process in the basic CMS configuration.

IV. Incorporation of Fault-Tolerant Distributed Computing Schemes into the Testbeds and Associated Instrumentation

With the MDN testbed and the CMS testbed established, several fault-tolerant distributed computing experiments have been conducted. As mentioned in Section E, the fault tolerance schemes that we have been focusing on are those intended for achieving system level fault tolerance in real-time applications. Experiments conducted to validate two such schemes will be sketched in this section in order to discuss the extensions made to the basic

testbeds to support the experiments. The purpose of the experiments was of course not only to validate the effectiveness of the schemes but also to study efficient implementation techniques for the schemes.

A. Experimental Validation of the Distributed Recovery Block (DRB) Scheme

1) Basic Principles of the DRB Scheme: The DRB (distributed recovery block) scheme is a technique for unified treatment of both hardware and software faults with minimal execution overhead [11], [12]. It provides efficient forward recovery in contrast to more time-consuming backward recovery such as rollback-and-retry. It is based on a combination of both the distributed processing and the recovery block structuring concepts. Recovery block (RB) [5], [18], consists of one or more routines, called try blocks, which are designed to produce the same or similar computation results, and the acceptance test (AT), which is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A try (i.e., execution of a try block) is thus always followed by an acceptance test. In a sense, RB is an enclosure of some recoverable activities of a process.

The real-time TCN's considered here are assumed to have the following characteristics:

1) A TCN consists of multiple computing stations, each executing one and only one RB.

2) The result produced from a computing station may become an input to another computing station or to the application environment.

3) A computing station may consist of one or more computing nodes. Multiple computing nodes within a computing station can be used either in a load-sharing or in a redundant processing mode.

The DRB scheme exploits concurrent execution of try blocks to facilitate fast forward recovery. For simplicity, only two try blocks in an RB, the primary and the backup, are used to illustrate the DRB scheme. The specification of the maximum execution time allowed for each try block is an integral part of the DRB scheme. A try not completed within the time limit due to hardware faults or excessive looping is treated as a failure. Therefore, the acceptance test can be viewed as a combination of both logic and time acceptance tests.

The DRB scheme realized with two nodes is depicted in Fig. 6. Both primary and backup nodes contain the same acceptance test, consisting of logic and time tests, and the same set of try blocks, *A* and *B*. However, the roles of the two try blocks are assigned differently in the two nodes. Primary node *X* uses try block *A* as the primary try block initially, whereas backup node *Y* uses try block *B* as the initial primary. Therefore, until a fault is detected, both nodes receive the same input data, process the data by use of two different try blocks (i.e., block *A* on node *X* and block *B* on node *Y*), and check the results by use of the acceptance test. Both nodes perform all these tasks concurrently The time acceptance test is used to ensure

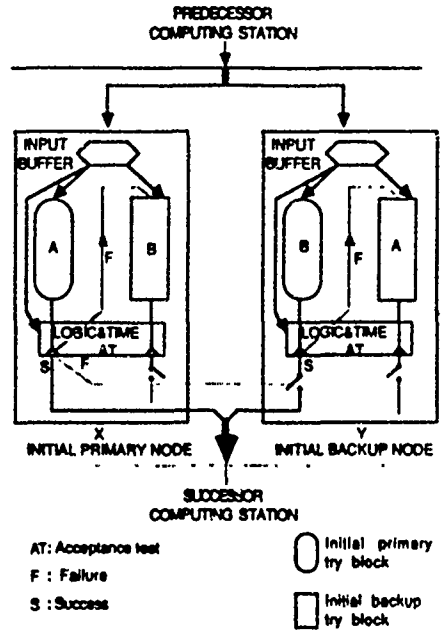


Fig. 6. The basic structure of the distributed recovery block (DRB).

timely behavior of both nodes. Also, note in the figure that the data kept in the input buffer is available for referencing by the acceptance test.

In a fault-free situation, both nodes will pass the acceptance test with the results computed with their primary try blocks. In such a case, the primary node notifies the backup of its success in the acceptance test. Thereafter, only the primary node sends its output to the successor computing station. However, if the primary node fails and the backup node passes its test, the backup node assumes the role of the primary, i.e., the nodes exchange their roles. To be more specific, the primary node attempts to inform the backup node upon its failure in passing the acceptance test. The backup node will take over the role of the primary as soon as it receives the notice. If the primary node is completely lost, the backup node will recognize the failure of the primary upon expiration of the preset time limit. It will then become the new primary. Since these interactions between two nodes are done asynchronously, the scheme does not suffer from synchronization overhead. On the other hand, if the backup node fails first, the primary node need not be disturbed. In both cases, the failed node attempts to become a new backup node; it first attempts to bring its application computation state or local database up-to-date by making a rollback and retry with its alternate try block. This attempt does not disturb the primary node.

2) Virtual Machine Mechanisms and Processes Supporting the DRB Scheme: In order to support experimentation of the DRB scheme on the MDN, the ECPM was extended with the following capabilities. First of all, the capability of establishing recovery points and making rollbacks was incorporated to support the attempt of a failed node to become a backup node. Secondly, the capability of detecting some exception conditions, e.g., overflow, and subsequently handling the situations in the

same way as acceptance test failures was incorporated. Thirdly, the capability of checking the acceptance test results of the partner node as well as detecting the timeout condition of the partner node was incorporated.

Fig. 2 shows an MDN configuration with the DRB scheme incorporated into nodes 3 and 6. The processes on both nodes as well as those on the adjacent nodes (1 and 4) were extended with the fault detection and recovery functions which facilitate the DRB scheme by use of the ECPM mechanisms mentioned above. Fig. 2 also shows an instrumentation made to the MDN in order to support measurement of the execution overhead caused by the introduction of the DRB scheme. Four "observation points" were established in the network. When a data set arrives at the designated observation point in the network, the node stamps the real-time and saves a copy of the time-stamped data in its local memory. When enough measured data are obtained, the time-stamped data are transferred to another computer system for data analysis. The ECPM was thus extended with a time-stamping and saving primitive.

In order to exercise the DRB mechanisms, the following types of faults were injected.

1) Total node failure: this was simulated by use of a node reset.

2) Software faults: infinite looping and arithmetic overflow were injected.

These faults were injected randomly rather than at preselected points in execution. Random injection is particularly important for validation of system-level fault tolerance schemes. When a node reset is forced, one does not know what execution stage the node is at, i.e., whether the node is in execution of the kernel, a system process, or an application process. Injection of transient hardware faults, e.g., transient faults of main memory, is in the plan for future work.

Fig. 7 illustrates the measurement results obtained. The curve connecting solid dots represents the time taken for a data set to "travel" from the point where it is picked up by one of the Analyzer-2 nodes in Fig. 2 (i.e., node 3 or node 6) to the point where the (processed) data set is stored into a buffer memory connected to the command scheduler node. Therefore, this curve corresponds to the case of using the DRB. On the other hand, the curve connecting small triangles represents the data set travel time in the case without the DRB. The gap between the two curves is the execution time increase, i.e., the overhead, due to the incorporation of the DRB. This overhead is mainly caused by the interaction between partner nodes, the establishment of recovery points, and the execution of the acceptance test. The average execution time increase shown in Fig. 7 is approximately 30 milliseconds. However, considering the inefficient implementation language (Extended Concurrent Pascal), and the slow processor (4 MHz Z8001) used, the amount of execution time increase shown in Fig. 7 is at least 20 times higher than that expected in the systems built with current off-the shelf hardware and software tools. For example, use of a processor running at 20 MHz will result in speedup by a factor of 5. Use of a more efficient language (an assembly language in the extreme case) will result in additional speedup by a factor of 4. Details of the results of the DRB experiment are referred to [3], [12].

A recent extension made to the DRB scheme is the capability of nondisruptive rejoin of repaired nodes. In other words, under the extended scheme called a repairable DRB scheme, a repaired node can be reincorporated into the previous DRB computing station. For example, if node 6 in Fig. 2 fails and is removed for repair, then node 3 will continue to operate without a backup partner. Later repaired node 6 can be reincorporated as the backup node without disrupting the real-time application. The main problem here is to bring the computation state or the local database of the rejoining node up-to-date, i.e., the newly introduced backup node (node 6) must import a copy of the database kept by the primary node (node 3) in order to prepare for serving as the backup node. This requires the cooperation of the primary node that has maintained the up-to-date database. The primary node exports a copy of its database using slack time, possibly through multiple control cycles. A simple case where the entire database could be copied during one control cycle was put to an experiment [21]. To support this, the ECPM was extended with a primitive used during the database duplication.

B. Experimental Validation of a Temporary Blackout (TB) Handling Scheme

1) Basic Principles of TB Handling: Temporary blackout (TB) handling means to continue to serve the needs of the real-time application environment while experiencing a temporary blackout that disrupts orderly operation of electronic components and erases the contents of registers and RAM's. Such a blackout may be caused by unreliable power sources or high energy events. In a uniprocess system, TB handling is a relatively simple problem. The system can periodically establish recovery points by saving the critical state variables into hardened nonvolatile storage and after experiencing a TB, conduct forward recovery by use of the saved state variables and the time information. The forward recovery procedure here is of course application-dependent.

In the case of a DCS, a new dimension of complexity is added to TB handling. Distributed processes must establish their recovery points in a coordinated manner; otherwise, the DCS that has experienced a TB may not be able to restore itself to a consistent state from which an attempt can be made for forward recovery. In addition, different processes may take different amounts of time for recovery. A simple approach here may be to make the recovered processes wait until all the cooperating processes have recovered before making a synchronous restart. However, it may be advantageous in many situations to allow processes to make an asynchronous restart. Designing distributed processes capable of performing

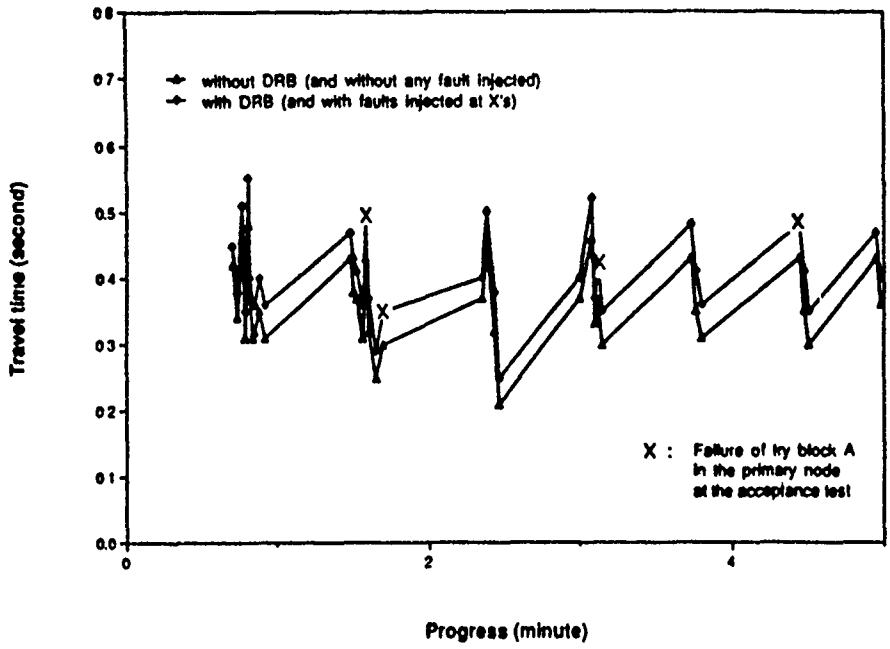


Fig. 7. Data travel time measured.

forward recovery followed by asynchronous restart is not yet a well understood subject.

2) Virtual Machine Mechanisms and Processes Supporting a TB Handling Scheme: A simple case of TB handling in a DCS has been experimented with in the DREAM Laboratory. The experiment was conducted on the CMS testbed running the multisensor control application depicted in Fig. 5. There are two factors in this testbed that make TB handling relatively simple.

1) The main control processor plays the role of a coordinator with respect to the overall application. Moreover, it was assumed that no processor would be completely lost during the experiment since the objective was to focus on the study of TB handling. Therefore, asynchronous restart of recovered processes was easy to implement because every recovered process was designed to merely check the health of its local sensor and controller until the main control process came alive.

2) The message traffic among the nodes was relatively low and at a reasonably steady rate. The CMS is equipped with a high-resolution global clock. These plus the coordinator-follower relationship existing between the main control processor and other nodes made it relatively simple to implement periodic establishment of distributed recovery points by use of the global clock.

A TB was simulated by use of a global interrupt capability present in the CMS. The capability called the "silver bullet" mechanism interrupts all the nodes in the CMS simultaneously when invoked. A spare node in the CMS was assigned the responsibility of generating a silver bullet. The CMS kernel was extended with a silver bullet handler that essentially kept reading the global clock until the period of time preset as the TB duration elapsed. The TB duration was made a parameter that could be set at the beginning of each experimental run. Once the TB period is over, then every node initiates the procedure of reloading saved critical variables and subsequent forward recovery. As a part of this procedure, the environment simulator resident in each node is reestablished as if no TB's had occurred. Another mechanism that had to be simulated to support the experiment was the hardened storage. A global memory module was designated as a hardened memory in which critical variables may be saved. These approaches of simulating a TB and a hardened memory have the advantage of creating controllable high-fidelity simulated events and environments.

Although the experimental setup has been fully operational for some time and some measured data have been collected, still further data need to be collected and substantial efforts to be spent for analysis of the data. Our study here is primarily focused on the following questions.

1) How difficult is it to implement an effective forward recovery logic?

2) How complex is the logic of TB handling?

3) How fast can the DCS recover after experiencing a TB?

4) How long a TB can the DCS endure without failing to meet the application requirements?

The experiences and the data obtained so far provide some very limited answers to these questions. In the case of the chosen application (Fig. 5), the forward recovery logic was not very complicated and proved to be effective. The amount of critical variables saved by each node was about 100 bytes on the average. The data obtained also indicated that in the case of the chosen application, recovery could be accomplished in less than 0.5 milliseconds when implemented with the off-the-shelf com

ponents. Much more experimental work on TB handling is needed to obtain adequate answers to the questions stated above.

C. Pros and Cons of the Testbed-Based Evaluation Approach

One advantage of this testbed-based evaluation approach is that relatively few assumptions about the execution environment are made. Truly distributed network configurations are used and real operating systems are run. Moreover, faults are injected in realistic manners and their real effects are demonstrated. Environment simulators are scaled-down real-time imitators of real environments and the application programs used are at least skeletons of the programs used in real operation. Therefore, compared to the results obtained from uniprocessor-based simulation approaches that use logical clocks rather than real-time clocks, the results from this testbed-based evaluation approach should be orders-of-magnitude more accurate in at least two aspects: timing and logical complexity. In other words, much more credible data can be obtained on entities such as recovery time, overhead execution time, logical complexity of implementing fault tolerance schemes, etc., from the testbed-based evaluation approach. Moreover, it enables obtaining useful estimates of the logical complexity that will be involved when the schemes are implemented into real application systems. A disadvantage is the relatively high level of efforts required to establish a real-time DCS testbed.

V. Desired Extensions

In the course of conducting the experiments mentioned in the preceding section and other experiments, several limitations of the testbed facilities in the DREAM laboratory have been recognized. Actually, removing some of the limitations identified seems to be beyond the state of the art. Nevertheless, even partial removal of these limitations will help accelerating advances in real-time distributed computing in general.

A. Instrumentation

The most obvious limitation was the lack of powerful debugging-aids, especially the tools that can provide snapshots of DCS states taken at various execution points of interest without perturbing the execution. An ideal tool would be what might be called a network single-stepping tool, which is capable of advancing the distributed nodes through one instruction followed by display of node states each time the user provides a keyboard stroke. Since different nodes generally execute different instructions at any given time, each single step really covers the actions of distributed nodes up to the time when one instruction is completed by a certain node. Unfortunately such a tool does not exist at present to this author's knowledge.

For constructing a capability of taking snapshots of DCS states, the global interrupt facility (e.g., silver bullet in the CMS) appears to be a useful component. It should be noted here that a global interrupt followed by synchronous restart will perturb the distributed execution to some extent since the global interrupt does not really stop the nodes at the same instant, but instead the nodes stop on completion of their current instructions.

B. Fault Injection

Simulation of a complete node failure by use of the node reset capability was an effective approach with respect to the randomness and freedom of insertion. However, a partially crippled node or an erratically behaving node cannot be created in such a manner. Although a software approach which, for instance, randomly erases some registers or RAM cells including those containing some program code, might be useful, it is limited in creating transient faults. Conceivable hardware approaches include warming circuit boards, surrounding circuits with electromagnetic fields, introducing electrical noises into buses, destabilizing power source and lines, etc.

C. Network Expansion

Both the MDN and the CMS offer limited numbers of nodes and limited connectivities among their nodes. At the same time they possess complementary characteristics. It became apparent that an integrated network offering more nodes and a mixture of internode connections, e.g., one including both the MDN and the CMS, would be a much more powerful testbed. Such an integrated network would significantly expand the range of DCS structures and applications that could be evaluated by use of it, compared to those possible with either the MDN or the CMS.

VI. Conclusion

A testbed-based approach to the evaluation of fault-tolerant distributed computing schemes has been discussed in this paper. Software structuring approaches that have evolved during the development of real-time tightly coupled distributed computing testbeds have been presented. Various virtual machine mechanisms and processes that effectively support real-time distributed computing and system-level fault tolerance schemes have also been presented. The testbed-based evaluation is an effective way of assuring the practical effectiveness of a complex scheme; the assurance obtained is a valuable addition to those that could be obtained through analytic evaluation or logical proof. Fault tolerance in real-time DCS's is a highly complex issue and numerous factors determine the effectiveness of the schemes. Therefore, a testbed-based evaluation of any new scheme is regarded as a very worthwhile effort. It is also becoming increasingly cost-effective as the costs of the hardware components of testbeds continue to decrease.

As discussed in this paper, the experiments that have been conducted with the testbeds established in the UCI DREAM laboratory have produced valuable data which could not have been produced through the approaches based on pure software simulators. Among other things, firm confidence in the practical effectiveness and the po-

tential performance of schemes such as the DRB, the TB handling, etc., have been obtained and useful insights into efficient implementation techniques obtained. Many more experiments including those aimed at evaluation of the schemes that are much more sophisticated than the DRB and the TB handling schemes (e.g., conversation scheme [8], [18]) are in the plan. However, the testbed structures discussed in this paper have limitations as discussed in Section V. It is hoped that new-generation testbeds without such limitations become available in the near future, thereby greatly easing the study of the design techniques for reliable real-time DCS's.

Acknowledgment

The author wishes to acknowledge the valuable help received from A. Abouelnaga, M. Beasley, C. Davis, E. Foudriat, N. Goddard, S. Heu, P. Hwang, V. Kobler, M. Kurtti, T. Lawrence, W. C. McDonald, E. B. Peterson, T. Smith, D. Thomas, A. Van Tilborg, C. R. Vick, N. Vosbury, H. Welch, S. M. Yang, J. C. Yoon, and J. H. You during the course of the work reported in this paper.

References

[1] B. Bhargava and J. Reidl, "The Raid distributed database system," *IEEE Trans. Software Eng.*, this issue, pp. 726-736.

[2] P. Brinch Hansen, *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

[3] W. Chu, K. H. Kim, and W. C. McDonald, "Testbed-based validation of design techniques for reliable distributed real-time systems," *Proc. IEEE (Special Issue on Distributed Databases)*, pp 649-667, May 1987.

[4] Honeywell Inc. Corp. Syst. Develop. Div., "Fault-tolerant distributed systems," Rome Air Develop. Center Contract F30602-85-C-0300, Final Rep., Dec. 1987.

[5] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," in *Lecture Notes in Computer Science*, vol. 16. New York: Springer-Verlag, 1974, pp. 171-187.

[6] IEEE Comput. Soc., *Summary of the Workshop on Design Principles for Experimental Distributed Systems*, Purdue Univ., Oct. 1986 (for the list of presentations made at this workshop, see *IEEE Comput. Soc. Distributed Processing Tech. Committee Newslett.*, vol. 10, no. 1, pp. 5-10, Mar. 1988).

[7] IEEE Comput. Soc., *Proc. Workshop Instrumentation for Distributed Computing Systems*, Sanibel Island, FL, Jan. 1987.

[8] K. H. Kim, "Approaches to mechanization of the conversation scheme based on monitor," *IEEE Trans. Software Eng.*, vol. SE-8, no. 3, pp. 189-197, May 1982.

[9] K. H. Kim, A. Abouelnaga, S. Heu, and S. M. Yang, "Process scheduling and prevention of communication deadlocks in an experimental microcomputer network," in *Proc. Real-Time Systems Symp.*, Dec. 1982, pp. 124-132.

[10] K. H. Kim, "Evolution of a virtual machine supporting fault-tolerant distributed processes at a research laboratory," in *Proc. 1st Int. Conf. Computer Data Engineering*, Apr. 1984, pp. 620-628.

[11] ——, "Distributed execution of recovery blocks: An approach to uniform treatment of hardware and software faults," in *Proc. 4th Int. Conf. Distributed Computing Sytems*, May 1984, pp. 526-532.

[12] K. H. Kim and H. O. Welch, "Distributed execution of recovery blocks: An approach to uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. Comput.*, vol. 38, May 1989.

[13] W. Kohler and B. P. Jeng, "Performance evaluation of integrated concurrency control and recovery algorithms using a distributed transaction processing testbed," in *Proc. 6th Int. Conf. Distributed Computing Systems*, May 1986, pp. 130-139.

[14] H. Kopetz and W. Merker, "The architecture of Mars," in *Proc. 15th Int. Symp. Fault-Tolerant Computing*, June 1985, pp. 274-279.

[15] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Trans. Comput.*, vol. C-36, pp. 933-940, Aug. 1987.

[16] W. C. McDonald and R. W. Smith, "A flexible distributed testbed for real-time applications," *Computer*, vol. 15, no. 10, pp. 25-39, Oct. 1982.

[17] W. C. McDonald and M. W. Beasley, "A real-time multi-microcomputer architecture employing a fully parallel crossbar switch," in *Proc. ICCD 83*, Oct. 1983, pp. 255-258.

[18] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220-232, June 1975.

[19] R. E. Schantz, R. H. Thomas, and G. Bono, "The architecture of the Cronus distributed operating system," in *Proc. 6th Int. Conf. Distributed Computing Systems*, May 1986, pp. 250-259.

[20] J. F. Shoch *et al.*, "Evolution of the Ethernet local computer network," *Computer*, vol. 15, pp. 10-27, Aug. 1982.

[21] J. C. Yoon, "An approach to design of fault-tolerant real-time tightly coupled networks and its experimental validation," Ph.D. dissertation, Dep. Comput. Sci. Eng., Univ. South Florida, May 1988.

K. H. (Kane) Kim (S'73-M'75-SM'86-F'89) received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1969, the M.A. degree in computer science from the University of Texas, Austin, in 1972, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1974.

From 1969 to 1971 he served as an officer in the Korean Army. From 1975 to 1986 he served on the faculty of the University of South Florida, Tampa, the State University of New York, Binghamton, and the University of Southern California, Los Angeles. While teaching at Binghamton, he also served as acting chairman of the Department of Computer Science for nine months. He is currently a Professor of Computer Engineering in the Department of Electrical Engineering at the University of California, Irvine. His current research interests are in the areas of reliable distributed processing and real-time software engineering. He is currently conducting both analytic and experimental research in the DREAM (Distributed Real-Time Ever Available Microcomputing) Laboratory.

Dr. Kim is a member of the Association for Computing Machinery and the IFIP Working Group 10.4, and a fellow of the IEEE Computer Society. He served as the Chairman of the IEEE Computer Society's Technical Committee on Distributed Processing from September 1984 to December 1986. In 1989 he plans to host the IEEE Computer Society's 9th International Conference on Distributed Computing Systems as the General Chairman.

Appendix A.XI

Consistency Constraints in Distributed Real Time Systems

CONSISTENCY CONSTRAINTS IN DISTRIBUTED REAL TIME SYSTEMS

H. Kopetz* and K. Kim**

**Institute f. Technische Informatik, Technical University of Vienna, Austria*
***University of California, Irvine, USA*

ABSTRACT

Real time information is invalidated by the passage of real time. In this paper a conceptual model of a distributed real time system is developed and a set of consistency constraints, concerning the time validity of real time information in distributed real time systems is presented. These consistency constraints concern the time interval between the observation of a control object and the use of this information by an application process.

Keywords. Real time systems, consistency of information, distributed system.

1. INTRODUCTION

A real time application consists of some equipment or plant and a controlling computer system. In the following, the equipment or plant will be called the control object, and the computer system will be called the real time control system or in short the real time system. The control object and the real time system have to cooperate closely to provide some service to an environment. This close interaction in the domain of time between the control object and the computer is characteristic for real time applications.

In most situations, the control object is structured such that short response time requirements can be allocated to dedicated processors /Fra81/. This is one reason for the general acceptance of distributed architectures in real time applications. Another reason is their potential for improved reliability and maintainability.

A distributed real time system can be decomposed into a set of autonomous computers, called nodes and a local area network between these nodes. There is no common memory available in such a loosely coupled distributed architecture. The communication and the synchronization between the nodes has to be realized solely by the exchange of messages across a serial communication link.

Since this message transmission from a node observing the control object to a node, which performs some processing, takes time and the validity of an observation is invalidated by the passage of time, the performance of the nodes and the speed of transmission have to match the timing requirements of the application. We say that the implementation has to satisfy certain consistency constraints concerning the validity of the information in the domain of real time.

It is the objective of this paper to present a set of consistency constraints for distributed real time systems. These consistency constraints will be specified in a conceptual model of a distributed real time application. Conceptual models are concerned with the semantics of an architecture, not with their syntactic appearance. Since time is the important element in any real time application, the conceptual model has to describe all relevant phenomena related to the progression of real time. The following section contains a short discussion of real time and states the model assumptions about the global time base. A more detailed discussion of this topic can be found in /Kop87/. Section 3 is concerned with the properties of the real-time-entities. Section 4 deals with the observation of rt-entities and introduces the concept of real time data objects. Section 5, which is the main section of this paper, presents the consistency constraints between the real time data objects and the associated rt-entity. Although we feel that some of these consistency constraints are fundamental to all distributed real time systems, others are application specific. It is up to a given implementation to provide the required mechanisms such that the specified consistency constraints can be met within the given fault hypothesis.

2. REAL TIME

The progression of real time from the past to the future can be represented by a directed timeline, the arrow of time. We call a point on the timeline an event and a section of the timeline between two events a duration or interval. The theory of time adopted in this model is based on a sequence of discrete time points--the numbered ticks of a reference clock--and

This work was supported in part by the Austrian-USA cooperative science progam under grant Nr P6010P and by the US Office of Naval Research Contract N N00014-87-K-0231

an equivalence interval between these time points /Her86/. Whenever an event is observed, the last tick number of the reference clock is taken as the time stamp of this event. The time interval between two ticks of the reference clock is called a "granule" of time. Since all events which occur within one granule have the same time stamp, the granularity of the reference clock limits the resolution of the time measurement.

In distributed systems a further effect has to be considered. Each node of the system contains its own real time clock, which cannot be fully synchronized with all other real time clocks of the system. No matter how good the synchronization of the clocks, there is always a small interval where one clock has ticked and another clock has not ticked yet. Whenever an event occurs during this small interval it will be measured by these two nodes with a tick difference. We call the maximum interval between two respective ticks of the clocks in the ensemble the accuracy of synchronization. It is evident that the accuracy of synchronization determines the smallest reasonable granularity of the time base.

In this model the progression of time is represented by an independent global time variable in each node of the system. We assume that this global time base, the global time, has the following properties

[1] The time base is chronoscopic, i.e. it does not contain any point of discontinuity.

[2] The metric of the time base is sufficiently close to the metric of an external time standard (e.g. TAI, see /AST84/).

[3] For any single event observed by any two nodes i and k of the distributed system

$$|t\_i(e) - t\_k(e)| \leq 1$$

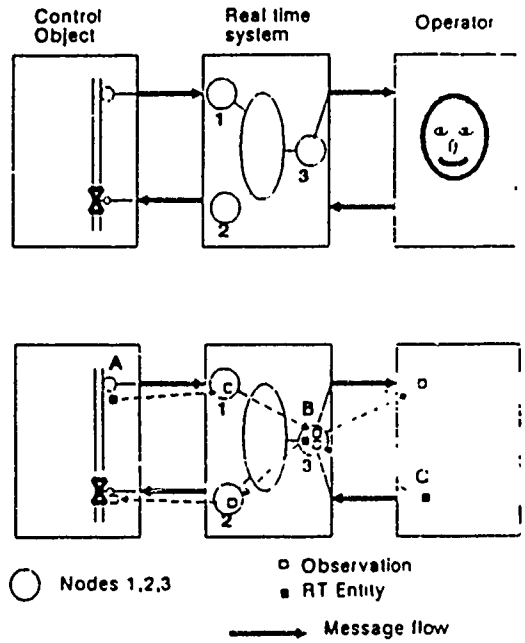
i.e. in a distributed system events can only be ordered in the domain of time if their time stamps are at least two ticks apart. Although it is theoretically appealing to assume a set of fully synchronized clocks, such a synchronization is technically not possible in a distributed system.

3. REAL TIME ENTITY

In any realtime application it is necessary to specify the entities, i.e. things and relations of interest, in order to describe the internal operation of the system. We introduce the neutral technical term "rt-entity" for this purpose. It is one of the crucial steps during system design to abstract from the wealth of information in the real world those rt-entities which have to be represented in the computer system

There is a set of attributes associated with every rt-entity. Some of these attributes are static, i.e. they do not change over the lifetime of the system, and some other are dynamic, i.e. they change as real time progresses. We call a dynamic attribute of a rt-entity a value and the set of all dynamic attributes of the rt-entity the value set. The name of a rt-entity is a special unique static attribute which makes it possible to refer to this rt-entity within the given context (name space). Other static attributes of rt-entities are the types and domains of values and the maximum speed of change of a value.

Let us provide some examples of rt-entities in a real time application. The actual position of a valve or the current pressure in a pipe are examples of rt-entities in the control object. The intended position of a valve is an example of a rt-entity in the computer system, while a desired setpoint for the pressure is an example of a rt-entity in the sphere of control of the operator (see also Fig. 1).



Every rt-entity is owned by a subsystem, i.e. this subsystem is free to set the rt-entity to any value within its domain. We say that the rt-entity is in the sphere of control /Dav79/ of this subsystem. The actual position of the valve or the current temperature of the vessel is in the sphere of control of the control object while the calculated position of the control valve is in the sphere of control of the computer system. A subsystem cannot modify any rt-entity outside its sphere of control. It can only observe such a rt-entity from the "outside".

In the computer system, any information about a rt-entity, will be represented in a data structure. We call a data structure which can accommodate the value set of a rt-entity a data object. If a given data object contains the information about a rt-entity which is owned by this subsystem, then this subsystem is free to set the value set of this data object within the given static constraints (e.g. data type). A data object which contains a rt-entity is called a primary data object. A data object which contains an observation of a rt-entity is called a secondary data object.

In a real time application there will be many rt-entities which are outside the sphere of control of the computer system, e.g. the pressure in a pipe or the setpoint selected by the operator. Within the computer system, there will be no primary data objects

the process will assume that the information displayed is a correct and timely image of the state of the process. We call such an assertion about current and archival observations and data objects and the global time a consistency constraint /Esw76, Tra82/.

In the following section consistency constraints for distributed realtime systems are presented. We feel that some of these consistency constraints are fundamental, i.e. they apply to a majority of real time systems and have to be satisfied at every moment in the lifetime of the system, while others are specific to particular applications.

During the analysis of a real time application the important consistency constraints must be specified. It is up to the implementation to provide the appropriate mechanisms such that the specified consistency constraints will be satisfied under all hypothesized (and specified) operating and fault conditions. If the behaviour of the real time environment is not in accordance with the hypothesized operating and fault conditions and, as a consequence, a catastrophic system failure results, then the responsibility for this failure is outside the realm of the system architect.

Fundamental consistency constraints

As indicated before, fundamental consistency constraints must be satisfied in the majority of real time system applications. A good architecture of a distributed real time system will support these fundamental consistency constraints at the operating system level or below, in order to simplify the application software.

(1) A current data object may never contain an archival observation.

An action or an operator reading a current data object must not access an observation which has lost its validity because of the passage of real time. This fundamental consistency constraint must be satisfied under all hypothesized load and failure conditions. If, in the before mentioned example about the AGV, the action which moves the vehicle into an intersection is based on an outdated traffic light observation, a catastrophe may result.

(2) In a current data object, the null value may per[illegible] the system specification.

[illegible] consistency constraint guarantees that any remote current data object is updated with a current observation at least x timeunits after the "old" observation has lost its validity.

(3) The time interval between a left event (t_l) of a discrete version of a rt-entity and the next point of observation (t_o) must be less than x timeunits, where x is contained in the specification.

This consistency constraint guarantees a timely observation of a state change in the environment. It determines the maximum duration of the period of observation in a periodic system.

(4) A data object in a subsystem which does not own the associated rt-entity may only be updated by a message from a subsystem which owns this rt-entity.

This consistency constraint guarantees the integrity of an observation in a remote data object.

Other consistency constraints:

(5) At any point of use, all current data objects which refer to the same rt-entity must contain the same observation.

This consistency constraint guarantees the internal consistency of a distributed system. Whenever there is a change from one version to the next version the use of the new version must be inhibited until all data objects are updated. There is a conflict between promptness of an update and consistency of an update. In some application the promptness of an update may be more important than the internal consistency.

(6) At any point of use , all current data objects which refer to the same rt-entity must contain the same observation or the null value.

This is a relaxation of consistency constraint (5) It is in the responsibility of the application software to decide what to do if it accesses a null value.

Although the presented consistency constraints may seem evident to the user of a real time system, they may pose a challenge to the system designer

6. CONCLUSIONS

In real time systems correctness and timeliness of information are of equal importance. Correct and timely information forms the basis for correct and timely actions and transactions. In this paper a conceptual model for the specification of the timing properties of information in distributed real time systems has been presented and set of consistency constraints has been developed. Any implementation of a distributed real time system must guarantee that the specified assertions about the validity of real time information are satisfied under all anticipated load [illegible] be reduced if appropriate mechanisms for the support of the consistency constraints are provided by the [illegible] research project MARS /Kop85/ we are developing a prototype of a real time operating system which will provide such a support.

REFERENCES

/AST84/Astronomical Almanac for 1984, Washington, London 1984

/Dav79/Davies, J.T., Data Processing Integrity, in Computing Systems Reliability, ed T Anderdson and B Randell, Cambridge University Press, London 1979, p.288 - 354

depends on the dynamics of the particular application.

One of the fundamental decisions in the design of a real time system has to be concerned with the determination of this validity interval of an observation.

Consider, for instance an AGV (automatic guided vehicle) before an intersection with a traffic light. This vehicle observes the traffic light at t_o and uses this observation sometimes later at t_u in order to decide if it safe to enter the intersection. The observation "the traffic light is green", which is used by the AGV at t_u, will become invalid as soon as the traffic light has changed to red. If our application is based on the assumption that the observation "the traffic light is green" is valid until the observation "the traffic light is red" has arrived at the user, then we require an absolutely reliable and timely transmission of this latter message. Any delay of the message "the traffic light is red" (caused by time redundancy in a low level transmission protocol) can lead to a catastrophe. It is unrealistic to assume the absolutely reliable and timely delivery of all messages in a distributed system.

In our opinion it is therefore more reasonable to include a validity time t_val as an atomic attribute of any information. This validity time is based on a commitment by the rt-entity and the observing subsystem that it is safe to use this observation until t_val. In the above example of the AGV the validity interval <t_val - t_o> will be determined by the duration of the yellow phase of the traffic light.

If every observation contains a validity time, then the requirement of an absolutely reliable and timely transmission of every message can be relaxed.

The parameter t_val of an observation is determined by the dynamics of the particular application. Since in a first approximation the analogue value of an attribute at the time t_val is at worst

$$v(t\_val) = v(t\_o) + g * (t\_val - t\_o)$$

the tolerated deviation of a value determines the termination point of the validity of an observation. The termination point t_val of an observation of a discrete version depends on the parameters d_min and d_min, as the previous example has shown.

The start time t_start

Let us assume that the same observation message is transmitted to many different users (nodes) in a distributed system. All these nodes are involved in a coordinated action. During the time interval between the arrival of this observation message at the first user and the last user, these users will operate on different versions of this message. This may violate some consistency constraint. In order to avoid such a consistency constraint violation a start event t_start is included in the observation. An observation may only be used after this point in time.

The value of t_start will normally be determined by the longest possible transmission delay of a message. In a system based on periodic observations, the three points in time, t_o, t_start and t_val can be coordinated, such that at any point in time there is either a consistent view by all nodes or a node knows that a message has been lost and can enter some exception handling procedure

State of an observation

An observation of a rt-entity is current at t_global if

$$t\_start \leq t\_global < t\_val$$

During the time interval

$$t\_o \leq t\_global < t\_start$$

an observation is called infant If

$$t\_global > t\_val$$

then the observation is archival At any point in time a rt-entity can have many archival observations

As soon as an observation reaches a user it is stored in a secondary data object. Data objects are the inputs and outputs of actions, i e primitive operations of the system Whenever an action reads a data object it must assume that certain assumptions about the contents of the data objects are satisfied A data object is current if it is intended to contain either a rt-entity, a current observation of a rt-entity or the null value. The set of all current data objects forms the real time data base of the real time system The archival data base is formed by an abstraction over the set of all archival observations

An example

Consider a simple real time application consisting of a pipe with a control valve, a pressure sensor and a controlling computer system (Fig 1) It is the objective of the system to set the control valve in a position such that a pressure selected by the operator is maintained

In this simple system we have three rt-entities the measured value A, the intended control valve position B and the setpoint for the pressure C The measured value is in the sphere of control of the control object, the intended control valve position is in the sphere of control of the real time computer system and the setpoint is in the sphere of control of the

A and transmits it to node number 3 and to the operator Node number 3 observes the setpoint C selected by the operator and the intended position of the control valve and transmits this position via node 2 to the control object While there is only one secondary data object in node 1 and 2, node 3 contains two secondary data objects (the observations of the measured values A and the setpoint C) and one primary data object (the rt-entity B denoting the intended valve position) It is evident that the quality of this simple control loop depends on the 'age' of the relevant observations

5. CONSISTENCY CONSTRAINTS

A real time system is used under the assumption that certain consistency assertions are satisfied An operator looking at his CRT console with a picture of

the process will assume that the information displayed is a correct and timely image of the state of the process. We call such an assertion about current and archival observations and data objects and the global time a consistency constraint /Esw76, Tra82/.

In the following section consistency constraints for distributed realtime systems are presented. We feel that some of these consistency constraints are fundamental, i.e. they apply to a majority of real time systems and have to be satisfied at every moment in the lifetime of the system, while others are specific to particular applications.

During the analysis of a real time application the important consistency constraints must be specified. It is up to the implementation to provide the appropriate mechanisms such that the specified consistency constraints will be satisfied under all hypothesized (and specified) operating and fault conditions. If the behaviour of the real time environment is not in accordance with the hypothesized operating and fault conditions and, as a consequence, a catastrophic system failure results, then the responsibility for this failure is outside the realm of the system architect.

Fundamental consistency constraints

As indicated before, fundamental consistency constraints must be satisfied in the majority of real time system applications. A good architecture of a distributed real time system will support these fundamental consistency constraints at the operating system level or below, in order to simplify the application software.

(1) A current data object may never contain an archival observation.

An action or an operator reading a current data object must not access an observation which has lost its validity because of the passage of real time. This fundamental consistency constraint must be satisfied under all hypothesized load and failure conditions. If, in the before mentioned example about the AGV, the action which moves the vehicle into an intersection is based on an outdated traffic light observation, a catastrophe may result.

(2) In a current data object, the null value may pertain [illegible] the system specification.

[illegible] consistency constraint guarantees that any remote current data object is updated with a current observation at least x timeunits after the "old" observation has lost its validity.

(3) The time interval between a left event (t_l) of a discrete version of a rt-entity and the next point of observation (t_o) must be less than x timeunits, where x is contained in the specification.

This consistency constraint guarantees a timely observation of a state change in the environment. It determines the maximum duration of the period of observation in a periodic system.

(4) A data object in a subsystem which does not own the associated rt-entity may only be updated by a message from a subsystem which owns this rt-entity.

This consistency constraint guarantees the integrity of an observation in a remote data object.

Other consistency constraints:

(5) At any point of use, all current data objects which refer to the same rt-entity must contain the same observation.

This consistency constraint guarantees the internal consistency of a distributed system. Whenever there is a change from one version to the next version the use of the new version must be inhibited until all data objects are updated. There is a conflict between promptness of an update and consistency of an update. In some application the promptness of an update may be more important than the internal consistency.

(6) At any point of use , all current data objects which refer to the same rt-entity must contain the same observation or the null value.

This is a relaxation of consistency constraint (5) It is in the responsibility of the application software to decide what to do if it accesses a null value.

Although the presented consistency constraints may seem evident to the user of a real time system, they may pose a challenge to the system designer

6. CONCLUSIONS

In real time systems correctness and timeliness of information are of equal importance. Correct and timely information forms the basis for correct and timely actions and transactions. In this paper a conceptual model for the specification of the timing properties of information in distributed real time systems has been presented and set of consistency constraints has been developed. Any implementation of a distributed real time system must guarantee that the specified assertions about the validity of real time information are satisfied under all anticipated load [illegible]

be reduced if appropriate mechanisms for the support of the consistency constraints are provided by the [illegible]

research project MARS /Kop85/ we are developing a prototype of a real time operating system which will provide such a support

REFERENCES

/AST84/Astronomical Almanac for 1984, Washington, London 1984

/Dav79/Davies, J T. Data Processing Integrity. In Computing Systems Reliability. ed T Anderdson and B Randell, Cambridge University Press, London 1979. p 288 - 354

/Esw76/Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L., The Notions of Consistency and Predicate Locks in a Database System., Comm. ACM, Vol. 9., No. 11, November 1976, p.624 - 633

/Fra81/Franta, W.R., Jensen, E.D., Kain, R.Y, Marshall, G.D, Real Time Distributed Computing Systems, Advances on Computers, Vol. 20 Academic Press, 1981, p.39 - 82,

/Her86/Herzog, R, Persistence of Information in Real Time Systems, Mars Report 4/86, Institut f. Technische Informatik, Technische Universitaet Wien, Vienna, Austria, February 1986

/Kop85/Kopetz, H., Merker, W., The Architecture of Mars, Proceedings of the 15th Symposium on Fault Tolerant Computing, Ann Arbor, Mich., IEEE Press, pp. 274-279, 1985

/Kop87/Kopetz, H., Ochsenreiter W., Clock Synchronization in Distributed Real Time Systems, IEEE Trans. on Computers, pp 933-940 August 1987

/Mer84/Merker, W., Die Behandlung der Zeit in verteilten PDV Systemen, Dissertation, Technische Universitaet Berlin, Germany, Nov. 1984

/Tra82/Traiger, I.L., Gray, J., Galtieri, C A., Lindsay, B.G., Transactions and Consistency in Distributed Database Systems, ACM Transactions on Database Systems, Vol.7, No. 3, Sept. 1982, p 323-342

DISCUSSION

Pimentel: What do you mean by "a data object can only be read but not be modified"?

Kopetz: A subsystem can uprate a data object only if it owns the associated real time entity (rt_entity)

Rodd: Could you give a brief summary of your experience so far with self-checking of components?

Kopetz: We have done some experimental evaluations of the self-checking behaviour of SBC's equipped with MC6800 s in respect to physical (hardware) faults and came to the following conclusions:

1. The probability of detecting permanent faults within a few milliseconds after occurrence is relatively high. The problem is the detection of transient faults of short duration (in the µsec range).

2. If you execute every task twice (even on the same hardware) at different times and compare the results (or a signature thereof), then in our experiments a very high degreee of error detection coverage of transients was achieved.

3. In the future, selfchecking concerns should be part of the hardware design. In the past few years many interesting techniques for the increase of the self-checking coverage of the hardware have been published.

DISTRIBUTED COMPUTER CONTROL SYSTEMS 1988

Proceedings of the Eighth IFAC Workshop
Vitznau, Switzerland, 13–15 September 1988

Edited by
M. G. RODD
Institute for Industrial Information Technology
University of Wales, UK
and
Th. LALIVE d'EPINAY
ABB Asea Brown Boveri AG
Mannheim FRG

Published for the
INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL
by
PERGAMON PRESS
OXFORD · NEW YORK · BEIJING · FRANKFURT
SÃO PAULO · SYDNEY · TOKYO · TORONTO

Appendix A.XII

Diagnosabilities of Hypercubes under the Pessimistic One-Step Diagnosis Strategy

nodes in the n-cube is the number of bit positions where the binary representations of the addresses of the two nodes differ from each other. A *path* in a hypercube is represented as a sequence of nodes in which every two consecutive nodes correspond to two processors directly connected to each other. A path is also represented as a sequence of connected edges, each representing an interprocessor link. The number of links on a path is called the *length of the path* in the hypercube. By the nature of the hypercube connectivity, the length of the shortest path between two nodes is the same as the Hamming distance between the two. The *node connectivity* $\kappa(G)$ of a grapn G is the minimum number of nodes whose removal results in a disconnected or trivial graph. A graph G is said to be *n-connected* if $\kappa(G) \geq n$. Whitney [19] has proven that a graph is n-connected if and only if there exist at least n disjoint paths between every pair of nodes in the graph. It was shown in [1] that an n-cube C has connectivity $\kappa(C) = n$.

Hakimi and Amin [8] obtained the following two conditions that are sufficient to assure that a system of N processors is precisely one-step t-fault diagnosable: 1) $N \geq 2t+1$, and 2) $\kappa(G) \geq t$ where $\kappa(G)$ is the node connectivity of the graph G representing the system. Since the n-cube C has connectivity $\kappa(C) = n$ and the number of processors in C, $N = 2^n$, satisfies the condition $N \geq 2n+1$, for $n \geq 3$, the n-cube (where $n \geq 3$) is precisely one-step n-fault diagnosable [1], [14].

In the rest of this paper, the following notation is also used. $\lfloor x \rfloor$ denotes the "floor function" of x, i.e., the largest integer not exceeding x, and $\lceil x \rceil$ denotes the "ceiling function" of x, i.e., the smallest integer not smaller than x.

III. Pessimistic One-Step Diagnosis of Hypercubes

A. Basic Properties of the Pessimistic Diagnosis Strategy

The main result presented in this section (about the degree of diagnosability of the n-cube under the pessimistic diagnosis strategy) is built upon some properties discovered earlier. These properties are summarized below.

A property that was useful in determining the degree of diagnosability of a system belonging to the special class of regularly structured systems, called the $D(N, t, X)$ class, was verified in [9]. It was shown that a $D(N, t, X)$ system is pessimistically one-step t'/t' fault diagnosable if and only if every pair of processors is tested by at least t' other processors. Later a generalization of this result that is applicable to an arbitrarily structured system was obtained in [3].

Theorem 1 [3]: Let $G(V, E)$ be the testing graph of a system S of N processors. Then S is pessimistically one-step t'/t' fault diagnosable if and only if for each integer p in the range $1 \leq p \leq t'$ and for each set of $2p$ nodes $V' \subseteq V$ (i.e., $|V'| = 2p$), the number of predecessor nodes (i.e., tester nodes) $|\mu^{-1}(V')| \geq t' - p + 1$. □

Therefore, the greatest t' that satisfies the condition stated in the above theorem is the degree of diagnosability of the system under the pessimistic diagnosis strategy.

B. The Degree of Diagnosability of the n-Cube under the Pessimistic Diagnosis Strategy

An important property regarding the connectivity of the n-cube that can be utilized in determining the degree of diagnosability is the following.

Theorem 2: In the n-cube where $n > 1$, any pair of processors can be tested by at least $2n - 2$ other processors. If the Hamming distance between two processors is either 1 or 2, then the number of other processors that can test the two processors is exactly $2n - 2$.

Proof: Consider two arbitrarily selected processors v_i and v_j which are separated by distance d (that is obviously in the range $1 \leq d \leq n$). According to a theorem proved in [13] there are n disjoint paths from processor v_i to processor v_j. More specifically, there are d paths of length d and $n-d$ paths of length $d+2$ from v_i to v_j. Thus, on each of $n-d$ paths there are at least two processors one of which is connected to processor v_i and the other connected to processor v_j. Therefore, on $(n-d)$ paths there are at least $2(n-d)$ processors of which at least $(n-d)$ processors can test processor v_i and at least $(n-d)$ other processors can test v_j. Also, on each of the other d paths there are 0, 1, or 2 processors which can test processor v_i or v_j or both, depending on whether d is equal to 1 or 2 or greater than 2. The total number of processors which can test processors v_i or v_j or both are thus as follows:

$$\begin{aligned} 2(n-d)+2d &= 2n && \text{for } d \geq 3, \\ 2(n-d)+d &= 2n-d && \text{for } d = 2, \text{ and} \\ 2(n-d) &= 2n-2d && \text{for } d = 1. \end{aligned}$$

Thus, for $d \leq 2$, $|\mu^{-1}\{v_i, v_j\}| = 2n - 2$ and for $d > 2$, $|\mu^{-1}\{v_i, v_j\}| = 2n$. □

Based on this theorem a proof is provided below that given a faulty n-cube with the fault bound of $2n-2$, where $n \geq 4$, all faulty processors can be removed by replacing at most $2n-2$ processors.

Theorem 3: The degree of diagnosability of the n-cube under the pessimistic one-step t'/t' fault diagnosis strategy, where $n \geq 4$, is $2n-2$.

Proof: Let $G(V, E)$ be the testing graph of the n-cube where $n \geq 4$. The goal here is to show that the condition stated in Theorem 1 is satisfied, i.e., for any integer p in the range $1 \leq p \leq t'$ and for each set of $2p$ nodes $V' \subseteq V$ (i.e., $|V'| = 2p$), the number of predecessor nodes $|\mu^{-1}V'| \geq t' - p + 1$. When $p = 1$, the number of nodes in V' is 2 and the condition becomes $|\mu^{-1}V'| \geq t'$. From Theorem 2 $|\mu^{-1}V'| \geq 2n - 2$. Thus, the condition of Theorem 1 is satisfied; that is,

$$t' \leq 2n - 2. \tag{1}$$

Let us now consider the case where $p > 1$. Let $V' = \{v_0, v_1, v_2, \cdots, v_{2p-1}\}$ denote the set of nodes in V' and $\{a_0, a_1, \cdots, a_{2p-1}\}$ denote the set of addresses of the corresponding nodes in V'. Without loss of generality, we can assume that $a_0 < a_1 < \cdots < a_{2p-1}$. We can also assume that the address of the first node $a_0 = 0$ and the address of the $(p+1)$th node $a_p \leq \lfloor (2^n - 1)/2 \rfloor$; since $G(V, E)$ has a symmetric structure, the nodes can always be renumbered to meet this condition. Therefore, the binary representation of a_i, $0 \leq i \leq p$, has "0" in its most significant bit position. Let us now consider another set of $p-1$ nodes $W = \{w_2, w_3, \cdots, w_p\}$ associated with the set of node addresses $\{a_2 + 2^{n-1}, a_3 + 2^{n-1}, \cdots, a_p + 2^{n-1}\}$. The binary representation of the address of w_i, $2 \leq i \leq p$, has "1" in its most significant bit position. In $G(V, E)$ there is no direct connection between $\{v_0, v_1\}$ and W because the binary representation of the address of v_0 or v_1 differs from the binary representation of the address of any w_i, $2 \leq i \leq p$, in at least two bit positions. That is,

$$\mu^{-1}\{v_0, v_1\} \cap W = \phi \tag{2}$$

where "$\cap$" denotes the set-intersection operator.

In addition, from Theorem 2,

$$|\mu^{-1}\{v_0, v_1\}| \geq 2n - 2. \tag{3}$$

Therefore,

$$|\mu^{-1}\{v_0, v_1\} \cup W| \geq 2n - 2 + p - 1.$$

Also, since the binary representation of the address of each w_i, $2 \leq i \leq p$, differs from the binary representation of the address of the corresponding node in V', i.e., v_i, in exactly one bit position, there is a direct connection between each w_i, $2 \leq i \leq p$, and v_i. That is,

$$\mu^{-1}\{v_2, v_3, \cdots, v_p\} \supseteq W. \tag{4}$$

From (2) and (4),

$$\begin{aligned} \mu^{-1}V' &= \mu^{-1}\{v_0, v_1\} \cup \mu^{-1}\{v_2, v_3, \cdots, v_p\} \\ &\quad \cup \mu^{-1}\{v_{p+1}, \cdots, v_{2p-1}\} - V' \\ &\supseteq \mu^{-1}\{v_0, v_1\} \cup W - \{v_2, v_3, \cdots, v_{2p-1}\}. \end{aligned}$$

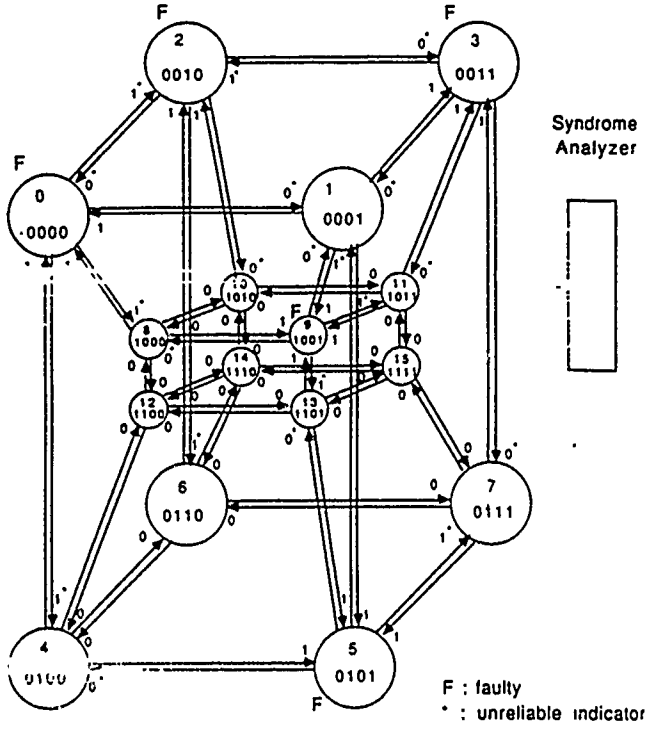


Fig. 1. A four-dimensional hypercube and a syndrome.

diagnosis strategy. Therefore, if the fault bound known or assumed exceeds the degree of diagnosability under the given diagnosis strategy, then the diagnosis strategy cannot be applied. The degree of diagnosability is a function of the testing connections (i.e., the set of "*tester–tested connections*") among the processors.

The original work in [16] was based on the repair strategy in which only those processors that were truly faulty were replaced. Therefore, the strategy may be called a *precise diagnosis strategy*. Later Friedman and Kavianpour [7], [9]–[11] proposed a strategy under which a set of r or fewer processors containing all faulty processors and possibly some processors of unknown status were identified and repaired. This strategy may be called a *pessimistic diagnosis strategy*. The motivating factor here is that under the precise diagnosis strategy, a situation where a good processor is completely surrounded by $t-1$ or less faulty processors, where t represents the fault bound, must not occur because the status of the isolated processor cannot be determined. Therefore, the degree of diagnosability under the precise strategy becomes low. Under the pessimistic strategy, such an isolated processor is treated as a potentially faulty processor and replaced. An important property common to both precise and pessimistic diagnosis strategies is that no faulty processors will remain undetected and unrepaired. While the pessimistic strategy might involve wasteful replacement of some operational processors, it has an advantage over the precise strategy in that the degree of diagnosability becomes higher.

The pessimistic diagnosis strategy under which up to t' processors may be replaced/repaired where t' is the fault bound, is called the (pessimistic) t'/t' diagnosis strategy [9]. It turns out that under the t'/t' strategy at most one fault-free processor may be replaced [3], [20]. In the remainder of this paper, the term pessimistic diagnosis refers to the pessimistic t'/t' one-step diagnosis. A procedure for finding minimum connection patterns among a given set of processor nodes that enable pessimistic diagnosis was developed in [15]. A procedure for identifying the processors to be repaired under the pessimistic diagnosis strategy was developed in [20]. Also an efficient (polynomial time) technique for determining the degree of diagnosability for a given multicomputer system under pessimistic diagnosis was developed in [18].

The degree of diagnosability of a given hypercube multicomputer system, in particular, under the pessimistic diagnosis strategy, is the subject of main concern in this paper. It has been known for quite some time that the degree of diagnosability of an n-cube under the precise strategy is n [1], [13]. This paper presents a proof that the degree of diagnosability of the n-cube, where $n \geq 4$. increases from n to $2n-2$ as the diagnosis strategy changes from the precise one-step strategy to the pessimistic one-step diagnosis strategy. (Although the algorithm in [18] can be used to find the degree of diagnosability of any individual hypercube, the closed form relation between an arbitrary-sized hypercube and its diagnosability does not follow directly from the algorithm.) When the fault bound adopted in an n-cube is the maximum number, i.e., $2n-2$, then the pessimistic diagnosis requires the use of all interprocessor links as testing links. However, it is shown here that if the fault bound is kept to the same number n in both cases of precise and pessimistic diagnosis, then the pessimistic strategy requires only $\lceil n/2 \rceil + 1$ testing links per processor whereas the precise strategy requires n testing links per processor. An algorithm for selecting $(\lceil n/2 \rceil + 1)*n/2$ bidirectional links in an n-cube for use as testing links is also presented.

In Section II, basic terminologies are introduced together with a graph model of a multicomputer system defined in [16] which is often called the PMC model. Section III then presents the results on the degree of diagnosability of the n-cube under the pessimistic one-step diagnosis strategy. The result on selection of testing links to realize the diagnosability of degree n is presented in Section IV. Section V provides a conclusion of the paper.

II. A Graph Model of a Multicomputer System and System-Level Diagnosis

In this paper, a multicomputer system is represented by a graph-theoretical model called the *testing graph* as in [16]. The testing graph is a digraph $G(V, E)$, where V is the set of nodes representing processors and E is the set of directed edges representing the *testing links* (i.e., the interprocessor links used as tester–tested connections) between the processors. Therefore, $(v_i, v_j) \in E$ if and only if v_i tests v_j. Associated with each processor $v_i \in V$ are the *tester set* $\mu^{-1}\{v_i\} = \{v_j : (v_j, v_i) \in E\}$ and the *tested set* $\mu\{v_i\} = \{v_j : (v_i, v_j) \in E\}$. Similarly, for $V' \subseteq V$, $\mu^{-1}\{V'\} = \{\bigcup_{v \in V'} \mu^{-1}\{v\}\} - V'$ and $\mu\{V'\} = \{\bigcup_{v \in V'} \mu\{v\}\} - V'$, where $\subseteq$ denotes the subset relationship and $\cup$ denotes the union operation. The outcome of a test in which processor v_i tests processor v_j is denoted by a_{ij}, and $a_{ij} = 1$ if processor v_i indicates that processor v_j is faulty whereas $a_{ij} = 0$ if processor v_i indicates that processor v_j is fault-free. If v_i is faulty, then outcome a_{ij} is unreliable. In the case of a hypercube, each internode link is bidirectional and thus can facilitate two testing links in opposing directions. A set of test outcomes of a multicomputer system that are analyzed together to determine faulty processors is called a *syndrome* of the system. Fig. 1 shows a syndrome resulting after a testing phase of a 4-cube with faulty processors P_0, P_2, P_3, P_5, and P_9. The figure also shows the presence of a syndrome analyzer that orders the processors to test others via testing links, collects the test results, and determines the set of nodes to be repaired. The connection between the syndrome analyzer and the processors of a hypercube may be either point to-point serial links or a multiaccess serial bus with the broadcast capability.

Definition 1 [16]. A system S is *precisely one-step t-fault diagnosable* if given the fault bound $t\,(>0)$, all the faulty processors in S can be correctly identified after a testing phase. □

Later Friedman and Kavianpour [7], [9], [10] defined t'/t' fault diagnosability as a part of introducing the concept of pessimistic diagnosis.

Definition 2 [9], [10]. A system S is *pessimistically one step t'/t' fault diagnosable* if given the fault bound $t'\,(>0)$ t' or fewer processors that include all the faulty processors present and possibly some processors of unknown status in S can be identified for replacement after a testing phase. □

Some of the basic graph-theoretic properties of the hypercube connection are now introduced. The *Hamming distance* d between two

Path 1:

$$\begin{aligned} x = (0x_{n-1}\cdots x_r\cdots x_1) &\dashrightarrow (0x_{n-1}\cdots \overline{x}_r\cdots x_1)\\ &\dashrightarrow (0x_{n-1}\cdots \overline{x}_{r+1}\overline{x}_r\cdots x_1)\cdots\\ &\dashrightarrow (0\overline{x}_{n-1}\cdots \overline{x}_r x_{r-1}\cdots x_1)\cdots\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r x_{r-1}\cdots x_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r x_{r-1}\cdots x_2\overline{x}_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r x_{r-1}\cdots x_3\overline{x}_2\overline{x}_1)\cdots\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r\cdots \overline{x}_1). \end{aligned}$$

Similarly, the second path is constructed by starting from address bit x_{r+1}. The $(n-r+1)$th path is constructed by starting from address bit x_n.

Path 2:

$$\begin{aligned} x = (0x_{n-1}\cdots x_r\cdots x_1)&\\ &\dashrightarrow (0x_{n-1}\cdots x_{r+1}\overline{x}_{r+1}x_r\cdots x_1)\\ &\dashrightarrow (0x_{n-1}\cdots x_{r+3}\overline{x}_{r+2}\overline{x}_{r+1}x_r\cdots x_1)\\ &\dashrightarrow (0\overline{x}_{n-1}\cdots \overline{x}_{r+1}x_r\cdots x_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_{r+1}x_r\cdots x_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_{r+1}x_r\cdots x_2\overline{x}_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_{r+1}x_r\cdots x_3\overline{x}_2\overline{x}_1)\cdots\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r\cdots \overline{x}_1). \end{aligned}$$

$\vdots$

Path $(n-r+1) = \lceil n/2\rceil + 1$:

$$\begin{aligned} x = (0x_{n-1}\cdots x_r\cdots x_1) &\dashrightarrow (1x_{n-1}\cdots x_r\cdots x_1)\\ &\dashrightarrow (1x_{n-1}\cdots x_r\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (1x_{n-1}\cdots x_{r+1}\overline{x}_r\cdots \overline{x}_1)\cdots\\ &\dashrightarrow (0x_{n-1}\cdots x_{r+1}\overline{x}_r\cdots \overline{x}_1)\\ &\dashrightarrow (0x_{n-1}\cdots x_{r+2}\overline{x}_{r+1}\overline{x}_r\cdots \overline{x}_1)\cdots\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r\cdots \overline{x}_1). \end{aligned}$$

Thus, there are $n-r+1 = \lceil n/2\rceil + 1$ disjoint paths.

Case 2) The distance between two processors, $f \le r$.

Consider two processors with addresses x and y differing in f positions. Without any loss of generality one can assume $x = (0\cdots 0x_f x_{f-1}\cdots x_1)$ and $y = (0\cdots 0y_f y_{f-1}\cdots y_1)$ where $x_i = \overline{y}_i$, $1 \le i \le f \le r$. The $(j+1)$th path, where $j = 0, 1, \cdots, (n-r)$, is constructed in three phases. Since $x_n = 0$, we are initially dealing with the part of the n-cube which was subject to Case 1 of Algorithm 1. In the first phase, address bits x_{n-j} and x_n are complemented one by one; for $j = 0$ only address bit x_n is complemented, of course. This path segment is available since no links in the segment were removed by application of Case 1 of Algorithm 1. In the second phase, address bits $x_1, \cdots, x_f$ are complemented one by one starting from x_1. This path segment is also available since no links in the segment were removed by application of Case 2 of Algorithm 1. In the third phase, address bits x_n and x_{n-j} are complemented one by one (for $j = 0$, only address bit x_n is complemented).

Path 1:

$$\begin{aligned} x = (0\cdots 0x_f\cdots x_1) &\dashrightarrow (10\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (10\cdots 0x_f\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (10\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 0\overline{x}_f\cdots \overline{x}_1). \end{aligned}$$

Path 2:

$$\begin{aligned} x = (0\cdots 0x_f\cdots x_1) &\dashrightarrow (010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (110\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (110\cdots 0x_f\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (110\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (01\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 0\overline{x}_f\cdots \overline{x}_1). \end{aligned}$$

Path 3:

$$\begin{aligned} x = (0\cdots 0x_f\cdots x_1) &\dashrightarrow (0010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (1010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (1010\cdots 0x_f\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (1010\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0010\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 0\overline{x}_f\cdots \overline{x}_1). \end{aligned}$$

$\vdots$

Path $(n-r+1) = \lceil n/2\rceil + 1$:

$$\begin{aligned} x = (0\cdots 0x_f\cdots x_1) &\dashrightarrow (0\cdots 010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (10\cdots 010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (10\cdots 010\cdots 0x_f\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (10\cdots 010\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 010\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 0\overline{x}_f\cdots \overline{x}_1). \end{aligned}$$

Therefore, there are $n-r+1 = \lceil n/2\rceil + 1$ disjoint paths.

Case 3) The distance between two processors $r < f < n$.

Let $x = (0\cdots 0x_f x_{f-1}\cdots x_1)$ and $y = (0\cdots 0y_f y_{f-1}\cdots y_1)$ be two address bits where $x_i = \overline{y}_i$, $1 \le i \le f$. The f bit positions can be divided into two groups: one consisting of r least significant bit positions and the other consisting of $f-r$ higher bit positions. By using the technique used in Case 1 above, $(f-r+1)$ disjoint paths can be constructed. (First, bit positions $x_{r+j}, \cdots, x_f$, where $0 \le j \le f-r$, are complemented, then x_n complemented, bit positions $x_1, \cdots, x_{r-1}$ complemented next, and x_n complemented last.) Also, by using the technique used in Case 2 above, $(n-f)$ disjoint paths can be constructed. (First, x_{n-j} and x_n, where $0 \le j \le n-f-1$, are complemented, second, bit positions $x_1, \cdots, x_r$ complemented, third, x_n complemented, $x_{r+1}, \cdots, x_f$ complemented next, and x_{n-j} complemented last.) Again there are $(f-r+1)+(n-f) = n-r+1 = \lceil n/2\rceil + 1$ disjoint paths.

Therefore, there are always $\lceil n/2\rceil + 1$ disjoint paths between p_x and p_y. □

Using Lemma 4, we can now prove the following useful property which is similar to that in Theorem 2.

Lemma 5: In an incomplete n-cube produced by Algorithm 1 where $n \ge 4$, any pair of processors can be tested by at least n other processors. If the Hamming distance between two processors is either 1 or 2, then the number of processors that can test the two processors is exactly n or $n+1$.

Proof: Consider an incomplete n-cube which is constructed by Algorithm 1. There are $\lceil n/2\rceil + 1$ disjoint paths between any pair of processors and each processor is tested by $\lceil n/2\rceil + 1$ other processors. Using the same logic as the proof of Theorem 2 if the Hamming distance between the two processors is d then each pair of processor is tested as follows:

$$\begin{cases} 2\lceil n/2\rceil + 2 & \text{for } d \ge 3,\\ 2\lceil n/2\rceil + 2 - d & \text{for } d = 2, \text{ and}\\ 2\lceil n/2\rceil + 2 - 2d & \text{for } d = 1. \end{cases}$$

□

From this and (3),

$$\begin{aligned}|\mu^{-1}V'| &\geq |\mu^{-1}\{v_0, v_1\} \cup W| - |\{v_2, v_3, \cdots, v_{2p-1}\}| \\ &\geq 2n - 2 + p - 1 - (2p - 2) \\ &= 2n - 2 - p + 1. \end{aligned} \quad (5)$$

If $t' \leq 2n - 2$. then from the condition (5),

$$|\mu^{-1}V'| \geq 2n - 2 - p + 1 \geq t' - p + 1.$$

That is, the condition in Theorem 1 is satisfied. In fact, even when t' is set to $2n - 2$ which is the maximum number meeting condition (1), the above condition is satisfied. However, if $t' = 2n - 2$, then n must be greater than 3 for the following reason. Consider the case where $p = t' = 2n-2$. Then the condition $|\mu^{-1}V'| \geq t'-p+1 = 1$ must hold true. This means that the total number of nodes in $G(V, E)$, $|V|$, must be in the following range.

$$|V| \geq |V'| + |\mu^{-1}V'| \geq 2p + 1.$$

So, $2^n \geq 2(2n - 2) + 1 = 4n - 3$.

In order to meet this condition, n must be greater than 3. To prove that the degree of diagnosability is exactly $2n - 2$, consider $t' = 2n - 1$ and $p = 1$. Then the condition would require $|\mu^{-1}V'| \geq 2n - 1$ for any two-element set V'. However, taking as V' any two neighbors it is obvious that $|\mu^{-1}V'| = 2n - 2$. Therefore, the degree of diagnosability of the n-cube, where $n \geq 4$, is $2n - 2$. □

Example 1: In a 4-cube, each of 2^4 processors can test four immediate neighbors and vice versa. According to Theorem 3, a 4-cube is pessimistically one-step 6/6 fault diagnosable. Therefore, if the fault bound is 6, we can repair a faulty 4-cube in one step by replacing at most six processors. In the 4-cube shown in Fig. 1, five processors 0, 2, 3, 5, and 9 are faulty. An analysis of the syndrome shown in the figure indicates that the five processors 0, 2, 3, 5, and 9 are definitely faulty and the status of processor 1 is unknown. Therefore, the system can be repaired by replacing six processors 0, 1, 2, 3, 5, and 9. Good processor 1 is replaced because it can be tested only by four other faulty processors and thus no reliable information about its status can be made available. Note that with a 4-cube the precise one-step diagnosis strategy can be used only when the fault bound is four or less. □

IV. Pessimistic One-Step Diagnosis of the n-Cube with Reduced Testing Connections

In the preceding section, it was shown that the n-cube ($n \geq 4$) could be diagnosed by use of the pessimistic one-step diagnosis strategy with the fault bound set to as high as $2n - 2$. When the fault bound adopted is the maximum number, i.e., $2n - 2$, then the pessimistic diagnosis requires the use of all interprocessor links as testing connections. That is, the number of testing connections used per processor is n. Similarly, when the precise one-step diagnosis strategy is used and when the fault bound adopted is the maximum number allowed under the strategy, i.e., n, the precise diagnosis requires the use of all interprocessor links as testing connections. However, if the fault bound is kept to the same number n in both cases of precise and pessimistic diagnosis, then the pessimistic strategy requires significantly fewer testing connections than the precise strategy does. In fact, the number of testing connections required under the pessimistic strategy is $\lceil n/2 \rceil + 1$ connections per processor.

In order to prove this, a method for constructing proper testing connection patterns will be given in the following. The key issue here is how to remove $\lfloor n/2 \rfloor - 1$ links per processor from an n-cube such that the remaining links can be used to facilitate $\lceil n/2 \rceil + 1$ testing connections per processor which enable the pessimistic one-step diagnosis of the n-cube under the fault bound of n.

Algorithm 1. Consider processor p_i in an n-cube where $n \geq 4$ and $0 \leq i \leq 2^n - 1$ and the binary address vector of p_i is denoted by $(i_n i_{n-1} \cdots i_1)$. Let r represent $\lfloor n/2 \rfloor$ for the sake of convenience.

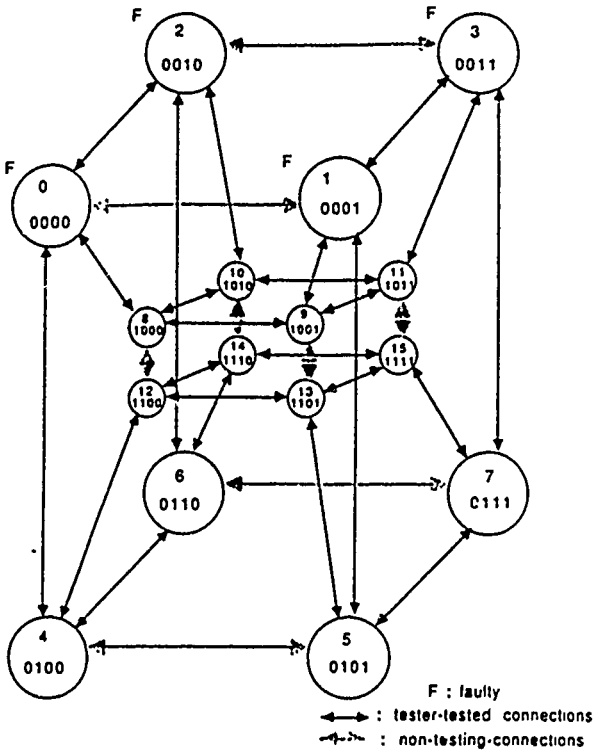


Fig. 2. A 4/4 fault diagnosable 4-cube that uses three testing connections per processor.

Case 1) $i < 2^{n-1}$: Remove the link from processor p_i to processor p_j if their address bits (i.e., binary address vector of p_i and p_j) differ only in one of the $(r - 1)$ least significant bit positions (i.e., the first, the second, $\cdots$, $(r - 1)$th bit positions).

Case 2) $i \geq 2^{n-1}$: Remove the link from processor p_i to processor p_j if their address bits differ in one of the following bit positions. $(n - r + 1)$th, or $(n - r + 2)$th, $\cdots$, or $(n - 1)$th bit positions. □

Example 2: Fig. 2 illustrates a 4-cube. According to Case 1 of Algorithm 1, links between pairs of processor (0, 1), (2, 3), (4, 5), and (6, 7) will not be used in testing since their address bits differ in the first bit position. Also according to Case 2 of Algorithm 1, links between pairs of processor (8, 12), (9, 13), (10, 14), and (11, 15) will not be used in testing since their address bits differ in the third position. □

Lemma 4: An incomplete n-cube produced by Algorithm 1, where $n \geq 4$, has connectivity $\lceil n/2 \rceil + 1$.

Proof: We will show that there are $\lceil n/2 \rceil + 1$ disjoint paths from each processor with address x to any other processor with address y. Let $x = (x_n x_{n-1} \cdots x_1)$ and $y = (y_n y_{n-1} \cdots y_1)$ be the binary address vectors. Without loss of generality we can assume $x_n = 0$ since an n-cube is symmetric. Also, let x_i represent the complement of the ith bit of $(x_n x_{n-1} \cdots x_i \cdots x_1)$.

Case 1) The distance between two processors is n.

The first path shown below is constructed in three phases. Since $x_n = 0$, we are initially dealing with the part of the n-cube which was subject to Case 1 of Algorithm 1. In the first phase, starting from x_r, address bits $x_r, \cdots, x_{n-1}$ are complemented one by one. This path segment is available since no links in the segment were removed by application of Case 1 of Algorithm 1. In the second phase, $x_n = 0$ is flopped. This link is available since links between the processors whose most significant address bits (i.e., x_n) differ but other address bits are identical were not removed by application of either case of Algorithm 1. We are now in the part of the n-cube which was subject to Case 2 of Algorithm 1. Then in the third phase, starting from x_1, address bits $x_1 \cdots x_{r-1}$ are complemented. This path segment is also available since no links in the segment were removed by the application of Case 2 of Algorithm 1.

Path 1:

$$\begin{aligned} x = (0x_{n-1}\cdots x_r\cdots x_1) &\dashrightarrow (0x_{n-1}\cdots \overline{x}_r\cdots x_1)\\ &\dashrightarrow (0x_{n-1}\cdots \overline{x}_{r+1}\overline{x}_r\cdots x_1)\cdots\\ &\dashrightarrow (0\overline{x}_{n-1}\cdots \overline{x}_r x_{r-1}\cdots x_1)\cdots\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r x_{r-1}\cdots x_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r x_{r-1}\cdots x_2\overline{x}_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r x_{r-1}\cdots x_3\overline{x}_2\overline{x}_1)\cdots\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r\cdots \overline{x}_1). \end{aligned}$$

Similarly, the second path is constructed by starting from address bit x_{r+1}. The $(n-r+1)$th path is constructed by starting from address bit x_n.

Path 2:

$$\begin{aligned} x = (0x_{n-1}\cdots x_r\cdots x_1)&\\ &\dashrightarrow (0x_{n-1}\cdots x_{r+1}\overline{x}_{r+1}x_r\cdots x_1)\\ &\dashrightarrow (0x_{n-1}\cdots x_{r+3}\overline{x}_{r+2}\overline{x}_{r+1}x_r\cdots x_1)\\ &\dashrightarrow (0\overline{x}_{n-1}\cdots \overline{x}_{r+1}x_r\cdots x_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_{r+1}x_r\cdots x_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_{r+1}x_r\cdots x_2\overline{x}_1)\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_{r+1}x_r\cdots x_3\overline{x}_2\overline{x}_1)\cdots\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r\cdots \overline{x}_1). \end{aligned}$$

⋮

Path $(n-r+1) = \lceil n/2\rceil + 1$:

$$\begin{aligned} x = (0x_{n-1}\cdots x_r\cdots x_1) &\dashrightarrow (1x_{n-1}\cdots x_r\cdots x_1)\\ &\dashrightarrow (1x_{n-1}\cdots x_r\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (1x_{n-1}\cdots x_{r+1}\overline{x}_r\cdots \overline{x}_1)\cdots\\ &\dashrightarrow (0x_{n-1}\cdots x_{r+1}\overline{x}_r\cdots \overline{x}_1)\\ &\dashrightarrow (0x_{n-1}\cdots x_{r+2}\overline{x}_{r+1}\overline{x}_r\cdots \overline{x}_1)\cdots\\ &\dashrightarrow (1\overline{x}_{n-1}\cdots \overline{x}_r\cdots \overline{x}_1). \end{aligned}$$

Thus, there are $n-r+1 = \lceil n/2\rceil + 1$ disjoint paths.

Case 2) The distance between two processors, $f \le r$.

Consider two processors with addresses x and y differing in f positions. Without any loss of generality one can assume $x = (0\cdots 0x_f x_{f-1}\cdots x_1)$ and $y = (0\cdots 0y_f y_{f-1}\cdots y_1)$ where $x_i = \overline{y}_i$, $1 \le i \le f \le r$. The $(j+1)$th path, where $j = 0, 1, \cdots, (n-r)$, is constructed in three phases. Since $x_n = 0$, we are initially dealing with the part of the n-cube which was subject to Case 1 of Algorithm 1. In the first phase, address bits x_{n-j} and x_n are complemented one by one; for $j = 0$ only address bit x_n is complemented, of course. This path segment is available since no links in the segment were removed by application of Case 1 of Algorithm 1. In the second phase, address bits $x_1, \cdots, x_f$ are complemented one by one starting from x_1. This path segment is also available since no links in the segment were removed by application of Case 2 of Algorithm 1. In the third phase, address bits x_n and x_{n-j} are complemented one by one (for $j = 0$, only address bit x_n is complemented).

Path 1:

$$\begin{aligned} x = (0\cdots 0x_f\cdots x_1) &\dashrightarrow (10\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (10\cdots 0x_f\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (10\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 0\overline{x}_f\cdots \overline{x}_1). \end{aligned}$$

Path 2:

$$\begin{aligned} x = (0\cdots 0x_f\cdots x_1) &\dashrightarrow (010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (110\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (110\cdots 0x_f\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (110\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (01\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 0\overline{x}_f\cdots \overline{x}_1). \end{aligned}$$

Path 3:

$$\begin{aligned} x = (0\cdots 0x_f\cdots x_1) &\dashrightarrow (0010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (1010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (1010\cdots 0x_f\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (1010\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0010\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 0\overline{x}_f\cdots \overline{x}_1). \end{aligned}$$

⋮

Path $(n-r+1) = \lceil n/2\rceil + 1$:

$$\begin{aligned} x = (0\cdots 0x_f\cdots x_1) &\dashrightarrow (0\cdots 010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (10\cdots 010\cdots 0x_f\cdots x_1)\\ &\dashrightarrow (10\cdots 010\cdots 0x_f\cdots x_2\overline{x}_1)\cdots\\ &\dashrightarrow (10\cdots 010\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 010\cdots 0\overline{x}_f\cdots \overline{x}_1)\\ &\dashrightarrow (0\cdots 0\overline{x}_f\cdots \overline{x}_1). \end{aligned}$$

Therefore, there are $n-r+1 = \lceil n/2\rceil + 1$ disjoint paths.

Case 3) The distance between two processors $r < f < n$.

Let $x = (0\cdots 0x_f x_{f-1}\cdots x_1)$ and $y = (0\cdots 0y_f y_{f-1}\cdots y_1)$ be two address bits where $x_i = \overline{y}_i$, $1 \le i \le f$. The f bit positions can be divided into two groups: one consisting of r least significant bit positions and the other consisting of $f-r$ higher bit positions. By using the technique used in Case 1 above, $(f-r+1)$ disjoint paths can be constructed. (First, bit positions $x_{r+j}, \cdots, x_f$, where $0 \le j \le f-r$, are complemented, then x_n complemented, bit positions $x_1, \cdots, x_{r-1}$ complemented next, and x_n complemented last.) Also, by using the technique used in Case 2 above, $(n-f)$ disjoint paths can be constructed. (First, x_{n-j} and x_n, where $0 \le j \le n-f-1$, are complemented, second, bit positions $x_1, \cdots, x_r$ complemented, third, x_n complemented, $x_{r+1}, \cdots, x_f$ complemented next, and x_{n-j} complemented last.) Again there are $(f-r+1)+(n-f) = n-r+1 = \lceil n/2\rceil + 1$ disjoint paths.

Therefore, there are always $\lceil n/2\rceil + 1$ disjoint paths between p_x and p_y. □

Using Lemma 4, we can now prove the following useful property which is similar to that in Theorem 2.

Lemma 5: In an incomplete n-cube produced by Algorithm 1 where $n \ge 4$, any pair of processors can be tested by at least n other processors. If the Hamming distance between two processors is either 1 or 2, then the number of processors that can test the two processors is exactly n or $n+1$.

Proof: Consider an incomplete n-cube which is constructed by Algorithm 1. There are $\lceil n/2\rceil + 1$ disjoint paths between any pair of processors and each processor is tested by $\lceil n/2\rceil + 1$ other processors. Using the same logic as the proof of Theorem 2 if the Hamming distance between the two processors is d then each pair of processor is tested as follows:

$$\begin{cases} 2\lceil n/2\rceil + 2 & \text{for } d \ge 3,\\ 2\lceil n/2\rceil + 2 - d & \text{for } d = 2, \text{ and}\\ 2\lceil n/2\rceil + 2 - 2d & \text{for } d = 1. \end{cases}$$

□

Theorem 6: An n-cube with the fault bound of n, where $n \geq 4$. can be diagnosed by the use of pessimistic one-step diagnosis and $k = \lceil n/2 \rceil + 1$ testing connections per processor.

Proof: By using Lemma 5 we can prove that the degree of diagnosability of an incomplete n-cube produced by Algorithm 1 under the pessimistic one-step diagnosis where $n \geq 4$. is n. The essence of this proof is not much different from the logic used in proving Theorem 3, and thus the details are omitted. Since an incomplete n-cube produced by Algorithm 1 uses $\lceil n/2 \rceil + 1$ testing connections per processor, the theorem is proved. □

Example 3: Fig. 2 illustrates a 4-cube in which $k = \lceil n/2 \rceil + 1 = 3$ interprocessor links emanating from each processor are used as testing connections. This 4-cube is obviously not precisely one-step 4 fault diagnosable, but it is pessimistically one-step 4/4 fault diagnosable. Through an exhaustive case analysis one can verify that the condition in Theorem 1 is satisfied by the testing connections in Fig. 2. For example, faulty processors 0, 1, 2, and 3 can be diagnosed under the pessimistic one-step strategy. □

A practical implication of Theorem 6 is that even if some links in the n-cube are broken, the one-step pessimistic diagnosis of the 2^n processors is possible as long as the fault bound does not exceed n. However, it should be noted that the problem of diagnosing arbitrary incomplete n-cubes is an open research problem.

V. Conclusion

The diagnostic power of the mutual testing based system diagnosis approach in diagnosing the n-cube has been the main subject of discussion in this paper. It has been shown exactly how much the degree of diagnosability of the n-cube increases as the diagnosis strategy changes from the precise one-step strategy to the pessimistic one-step strategy. The pessimistic one-step diagnosis appears to be a highly attractive strategy for use in hypercube systems, considering the high degree of diagnosability that it provides, the relatively small number of tester–tested connections that it uses, and the uselessness of a processor of unknown status surrounded completely by faulty processors in typical application environments.

The pessimistic diagnosis of the hypercube is a relatively young research subject. Many important questions such as the diagnosability of the hypercube with some broken links remain unanswered. Techniques for efficient implementation of the diagnosis strategies also need to be developed. Much more research, both analytic and experimental, is needed before this research field reaches a point where designers of highly reliable hypercube-based systems can draw much expected benefits.

References

[1] J. R. Armstrong and F. G. Gray, "Fault diagnosis in a Boolean n cube array of microprocessors," *IEEE Trans. Comput.*, vol. C-30, pp. 587–590, Aug. 1981.

[2] P. Banerjee *et al.*, "An evaluation of system-level fault tolerance on the intel hypercube multiprocessor," in *Proc. 18th Int. Symp. Fault-Tolerant Comput.*, 1988, pp. 362–367.

[3] K. Y. Chwa and S. L. Hakimi, "On fault identification in diagnosable systems," *IEEE Trans. Comput.*, vol. C-30, pp. 414–422, June 1981.

[4] A. T. Dahbura and G. M. Masson, "An $O(n^{2.5})$ fault identification algorithm for diagnosable systems," *IEEE Trans. Comput.*, vol. C-33, pp. 486–492, June 1984.

[5] A. T. Dahbura, "System-level diagnosis: A perspective for the third decade," Tech. Rep., AT&T Bell Labs., 1987.

[6] E. Dilger and E. Ammann, "System level self diagnosis in n cube connected multiprocessor networks," in *Proc. 14th Int. Symp. Fault-Tolerant Comput.*, 1984, pp. 184–189.

[7] A. D. Friedman, "A new measure of digital system diagnosis," in *Proc 5th Int. Symp. Fault-Tolerant Comput.*, 1975, pp. 167–170.

[8] S. L. Hakimi and A. T. Amin, "Characterization of connection assignment of diagnosable systems," *IEEE Trans. Comput.*, vol. C-23, pp. 86–88, Jan. 1974.

[9] A. Kavianpour, "Diagnosis of digital system using t/t measure," Ph.D. dissertation, Dep. EE-Systems, Univ. of Southern California, June 1978.

[10] A. Kavianpour and A. D. Friedman, "Efficient design of easily diagnosable systems," in *Proc. 3rd USA–JAPAN Computer Conf.*, Oct. 1978, pp. 251–257.

[11] ——, "Trade-offs in system level diagnosis of multiprocessor systems," in *Proc. AFIPS National Comput. Conf.*, July 1984, pp. 173–181.

[12] C. Kime, "System diagnosis," in *Fault-Tolerant Computing: Theory and Techniques*, D. K. Pradhan, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[13] J. Kuhl and S. Reddy, "Distributed fault-tolerance for large multiprocessor systems," in *Proc. 7th Int. Symp. Comput. Architecture*, 1980, pp. 23–30.

[14] ——, "Fault diagnosis in fully distributed systems," in *Proc. 11th Int. Symp. Fault-Tolerant Comput.*, 1981, pp. 100–105.

[15] N. Maxemchuk and A. Dahbura, "Optimal diagnosable system design using full difference triangles," *IEEE Trans. Comput.*, vol. C-35, pp. 837–839, Sept. 1986.

[16] F. P. Preparata, G. Metze, and R. T. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Trans. Electron. Comput.*, vol. EC-16, pp. 848–854, Dec. 1967.

[17] A. K. Somani, V. K. Agarwal, and D. Avis, "A generalized theory for system level diagnosis," *IEEE Trans. Comput.*, vol. C-36, pp. 538–546, May 1987.

[18] G. Sullivan, "A polynomial time algorithm for fault diagnosability," *IEEE Symp. Foundations Comput. Sci.*, 1984, pp. 148–156.

[19] H. Whitney, "Congruent graphs and the connectivity of graphs," *Amer. J. Math.*, vol. 54, pp. 150–168, 1932.

[20] C. L. Yang, G. M. Masson, and R. Leonetti, "On fault identification and isolation in t_1/t_1-diagnosable systems," *IEEE Trans. Comput.*, vol. C-35, pp. 639–644, July 1986.

A Hardware-Oriented Algorithm for Floating-Point Function Generation

E. Pearse O'Grady and Baek-Kyu Young

Abstract—An algorithm is presented for performing accurate, high-speed, floating point function generation for univariate functions defined at arbitrary breakpoints. Rapid identification of the breakpoint interval which includes the input argument is the key operation in the algorithm. A hardware implementation which makes extensive use of read/write memories illustrates the algorithm.

Index Terms—Floating-point computation, function generation, simulation, table lookup.

I. Introduction

Scientific computing applications often involve evaluation of continuous functions of one or more variables represented by empirically derived data sets. The values taken by these nonanalytic functions are known only at a set of discrete breakpoints. At points other than the breakpoints, function values must be approximated by interpolation (or extrapolation) with a suitable approximating function, a relatively time consuming process. As an example, over 60% of the total run time is devoted to function-evaluation operations in a simulation of a jet engine reported by O'Grady and Wang [7]. Clearly there is a

Manuscript received April 15, 1987; revised June 22, 1990. This work was supported in part by NASA under Grant NAG 3-113.

E. P. O'Grady is with the Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287.

B.-K. Young is with Motorola Microcomputer Division, Tempe, AZ 85282.

IEEE Log Number 9038767.

Appendix A.XIII

Real-Time Distributed Computing Testbeds Established

As a part of the experimental work, the PI has established a laboratory named the DREAM Laboratory. The laboratory was started in the PI's former institution, Univ. of South Florida, in 1980 and was moved to his current institution, Univ. of California, Irvine (UCI), in January 1987.

The DREAM Lab consists of the Loosely Coupled Network (LCN) Section and the Tightly Coupled Network (TCN) Section. There are two major testbeds established in each section. The equipment in the LCN Section includes a Cromemco LAN connecting four MC68000-based microcomputers and a LAN of four 80386-based PC's made by AST Research Inc. and connected by Ethernet. The Cromemco LAN is several years old and the PC LAN was established in 1990. These form the first testbed in the LCN Section. The operating system and real-time application software running on the Cromemco LAN has been almost fully transported to the PC LAN with some restructuring of the operating system. The second testbed was added in the Fall of 1989. A 3-node ADS (Autonomous Decentralized System) made by Hitachi was established. The ADS has a unique communication architecture called the data field that enables easy expansion and reconfiguration. Each node is based on M68020 and has a unique network interface. One node runs a UNIX-ACP combination and two other nodes run the ACP (Atom Control Program) which is a Hitachi's proprietary real-time operating system.

The TCN Section includes two major networks: (1) One called the Macro-Dataflow Network (MDN) is a homemade network of six single-board Z8001-based microcomputers connected through up to 12 two-port buffer memory modules and (2) the other called the Crossbar Multi-microcomputer System (CMS) is a network of seven single-board microcomputers and five multi-port shared memory modules connected through a crossbar connection subsystem and manufactured by the Unisys Corporation in Huntsville, Alabama.

Real-time distributed operating systems and distributed application programs have been developed to run on three computer networks in the DREAM Lab., i.e., PC LAN, MDN, and CMS, and a real-time application program was added to run under the manufacturer's operating system of the ADS. The application programs included in the two TCN testbeds and the PC/Cromemco testbed are the distributed real-time control programs combined with simulators of applications environments with sensor devices and actuators. The three testbeds deal with the three different types of real-time object tracking applications: (1) tracking with a ground-based radar (the MDN testbed), (2) tracking with a sensor boarded on a high-speed moving vehicle (the CMS testbed), and (3) cooperative tracking by sensors distributed over multiple satellites (the PC/Cromemco testbed, also called the Defense Satellite Network testbed). The PC/Cromemco

testbed thus deals with a WAN application although the hardware base used is a LAN. It is a kind of dual-purpose LAN/WAN testbed.

The three testbeds dealing with object tracking applications were used in conducting the fault tolerance experiments discussed in the main report. Figure 1 provides a brief summary of the current status of the three testbeds.

Quite a few software and hardware tools for prototyping of real-time computer networks have been established in the UCI DREAM Laboratory. They include operating system components, communication primitives, and high level languages such as C, Extended Concurrent Pascal, Modula-2, and Unisys PDL. There are also tools for measurement of message delays. An approach formulated for rapid prototyping of software for real-time computer networks is a two-step approach in which the first inefficient version is obtained with the aid of an abstract high level language such as Extended Concurrent Pascal or ADA, and then using the first version as a blueprint, the final version is written in an efficient language such as C. This approach has been partially tested in the DREAM Laboratory with good results. The main approach established in the DREAM Laboratory for network performance measurement is to install "observation points" within a network such that when a message passes through an observation point, a time-stamped record is made by an observing machine. By comparing the time-stamped records made at different observation points, the time taken for a message to travel between observation points can be obtained.

PC-based facilities for graphic display of the run-time status of both the distributed application software and the hardware configuration have been established as integral components of both testbeds (CMS and MDN) in the TCN Section.

Real-Time Defense Computer Network Testbeds Established

- Terminal defense controller (TDC) testbed

  - simulates ground-based radar tracking activities
  - built on the six-node tightly coupled microcomputer network
    called the MDN (Macro-Dataflow Network)
  - uses an OS developed in house
  - about 10K lines of Extedned Concurrent Pascal, C, and Z8001 assembly code

- On-board intelligence (OBI) testbed

  - simulates interceptor-borne optical sensor and data processor activities
  - built on the seven-node tightly coupled microcomputer network
    called the CMS (Crossbar Multi-microcomputer System)
  - uses an OS developed in house
  - about 5K lines of C and Z8001 assembly code

- Defense satellite network (DSN) testbed

  - simulates a squad of mid-course satellite-borne radar tracking processors
  - built initially on the four-node local area network of
    Cromemco Z2/68000 microcomputers
  - uses an OS developed in house
  - about 10K lines of Exended Concurrent Pascal, C, and M68000 assembly code
  - Entire software was first ported to a LAN of Intel 80386-based PC's in 1990.

Figure 1. An overview of the three real-time defense computer networks established in the UCI DREAM Lab.

Distribution List

Addressee	Copies
Scientific Officer: Dr. Gary M. Koob ONR, Code 1133 800 North Quincy Street Arlington, VA 22217	1
Administrative Contracting Officer: Mr. Robert L. Bachman Office of Naval Research Resident Representative University of California, San Diego Scripps Institution of Oceanography, A-034 La Jolla, CA 92093-0234	1
Director, Naval Research Laboratory Attn: Code 2627 Washington, D.C. 20375	6
Defense Technical Information Center Bldg. 5, Cameron Station Alexandria, VA 22314	12